User Co-scheduling for MPI+OpenMP Applications Using OpenMP Semantics

Antoine Capra[1(✉)], Patrick Carribault[3], Jean-Baptiste Besnard[1], Allen D. Malony[2], Marc Pérache[3], and Julien Jaeger[3]

[1] ParaTools SAS, Bruyeres-le-Chatel, France
{capra,jbbesnard}@paratools.com
[2] ParaTools Inc., Eugene, USA
malony@paratools.com
[3] CEA, DAM, DIF, 91297 Arpajon, France
{patrick.carribault,marc.perache,julien.jaeger}@cea.fr

Abstract. The evolution of parallel architectures towards machines with many-core processors and high node-level concurrency is putting an end to the pure-MPI programming model. Simulations codes must expose multiple levels of parallelisms inside and between nodes, combining different programming models (e.g., MPI+X), to productively use current and future supercomputers. MPI+OpenMP is a common hybridization approach. However, recent evolutions in the OpenMP standard presents options for how OpenMP tasking constructs might be used when mixing fine-grained computation and communications. Various approaches are discussed and compared in this context. Advantages and limitations of the approaches are detailed, including potential improvements to OpenMP in order ease both the integration and progress of MPI calls. These methods are applied to a representative stencil code and demonstrate improvements on the overall execution time as a result of more efficient mixing of MPI and OpenMP.

1 Introduction

Parallel scientific applications are designed to take advantage of the resources they are provided for execution. When considering current architectures, the optimization spectrum is wide, ranging from vectorization at a core level to distributed operations involving millions of cores. The present rise of many-core processors is shaping the spectrum further with greater node-level concurrency, resulting in less memory per thread of execution. Whereas a pure MPI model [5] has been adequate before, memory replication within a node and communication overhead across many threads is becoming problematic. Hybrid programming methods that combine a shared-memory model with a distributed-memory one are has now a compulsory avenue when it comes to writing efficient parallel code. When considering MPI+X hybridization, X = OpenMP has become the de-facto standard. In hybridizing legacy codes, it is most often the case that OpenMP is applied at loop level for intra-node parallelism [6], with MPI for

B.R. de Supinski et al. (Eds.): IWOMP 2017, LNCS 10468, pp. 203–216, 2017.
DOI: 10.1007/978-3-319-65578-9_14

inter-node communication. However, by separating MPI and OpenMP phases, parallelism is essentially bulk-synchronous, alternating between communication and computation phases. In such a model, communications are done by a single thread, creating a loss of parallelism combined with extra fork-join overheads. Thus, despite being a practical approach, secluding MPI and OpenMP from each other will expose performance factors that eventually prevent the program from scaling.

We propose to rethink MPI+X hybridization with respect to their runtime requirements and flexibility for closer mixing of models. In particular, we are interested in how a program written with MPI+X in mind can express fine-grained parallelism and communication through OpenMP. Given the new features introduced in the OpenMP standard for programs to invoke MPI functions inside parallel regions, the opportunity is there for mitigating the bulk-synchronous nature of most MPI+OpenMP applications. Our work focuses on OpenMP tasks and presents an approach for hybrid tasking patterns that can be more performant. In this process, we observe some limitations in existing OpenMP runtimes and propose extensions to OpenMP oriented towards runtime stacking.

By considering a task-based model, the expression of both MPI and computation phases is more natural. Iterations are seen as a directed graph mixing MPI and compute tasks. Tasks are vertexes in the graphs and edges represent dependencies between tasks. This leads to the expression of an MPI+OpenMP program as a Directed Acyclic Graph (DAG). One benefit of a DAG representation is that finer-grained parallelism is more exposed, as are the dependencies and critical paths that constrain performance.

To demonstrate our approach, we focus on the critical path arising in DAGs representing stencil-based computations, including spatial dependencies. In particular, our goal is to reduce the coupling arising from communications between distributed memory regions, by identifying as soon as possible those parts of the computation where dependencies were satisfied. In this formalism, MPI tasks are the one leading to the highest parallel overhead, possibly delaying computation. From this starting point, it is shown how tasking patterns can mitigate communication impact, giving a higher priority to MPI tasks and splitting computational border into multiple regions – eventually moving communications inside the parallel region.

In the rest of the paper, we first describe task support in the context of OpenMP runtimes and discuss how it is beneficial to the expression of hybrid computation. After exploring various alternatives, we present an approach leveraging OpenMP tasks with dependencies to mix MPI and computation. The approach is validated using the stencil benchmark, demonstrating the impact of communication progress on the overlaps. We then present potentials improvements to the OpenMP standard for model mixing when applying the tasking model. Other research have contributed to our ideas and we give an overview of this related work. The paper concludes with future prospects to pursue.

2 OpenMP Tasking

OpenMP's origins began with loop-level parallelization, but over time an increasing variety of parallel constructs have been proposed for adoption in the OpenMP standard. One of the main drawbacks of parallel execution only in loops is that it breaks the program (within a node) into sequential and parallel regions. There is also the fact that not all loops are easily parallelized. Some may have complex dependencies and others may rely on external sequential (not thread-safe) libraries. Or it might simply be that time has not be taken to rewrite the loop code properly to enable parallel execution. In any case, limiting parallelism to just loop regions can constrain the performance gain in an OpenMP program. The well-known Amdahl Law states that the sequential part of a parallel code will bound its strong scaling speedup. For example, if 20% of the time an OpenMP program executes in a sequential region, maximum speedup is 5, even under the assumption of 100% efficiency in parallel loop execution. Thus, it is crucial to consider how OpenMP can express parallelism beyond loops.

To this end, the concept of tasks is being considered by parallel programmers to improve the scalability of their applications. OpenMP did not provide task until Version 3. At this point, `task` and `taskwait` are defined and the `barrier` is a scheduling point for tied tasks. OpenMP v4 introduced `taskgroups` to allows more abstraction and hierarchy, with `depend` being used to express dependencies and explicit scheduling points for untied tasks removed. The latest version of the OpenMP standard (v4.5) adopted `taskloop` and priorities. With these tasks constructs and their functional and runtime support, OpenMP now provides a way of defining parallel execution at a fine-grained level.

OpenMP tasking will notably enhance the opportunities for shared-memory parallelism and efficiency. Consequently, tasking capabilities also afford us a path to develop hybrid (MPI+OpenMP) codes with better performance than previously obtained.

3 Hybrid Alternatives

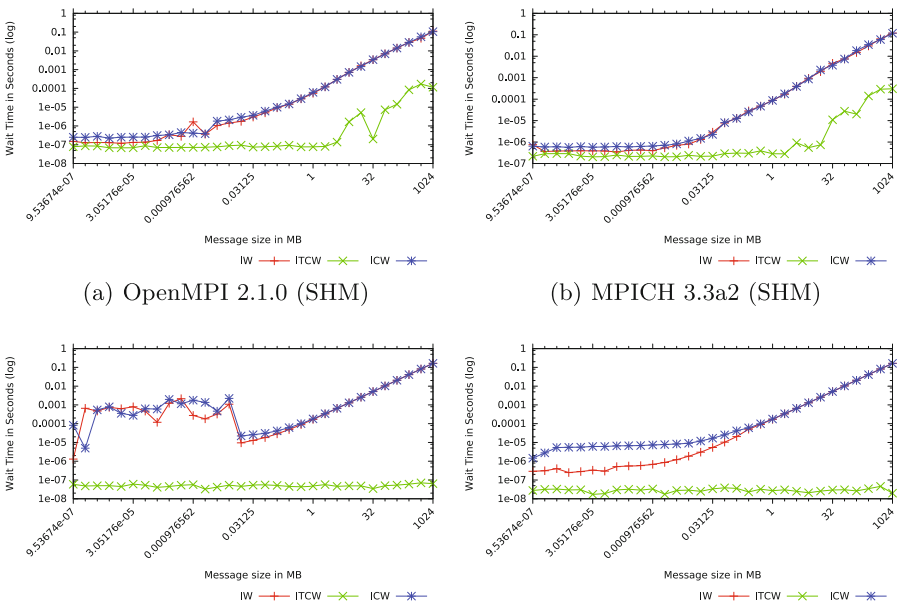
When mixing MPI and OpenMP one crucial aspect is how runtimes are going to interoperate. Because the MPI runtime is managing communications, it is by definition not performing computational work. While MPI asynchronous communication allows for the overlap of communication and computation, a main interest of hybridization is to enable node-level parallelism in a manner whereby the OpenMP runtime more efficiently interfaces with communication operations. To better describe our approach, we present three different MPI scenarios and reason about the performance costs involved.

In this Section we consider the `MPI_Irecv` and `MPI_Isend` calls. These functions allow for the posting of an asynchronous message. Both functions create an `MPI_Request` which can be used to either wait for the communication completion with `MPI_Wait` or test for its completion with `MPI_Test`. Using these calls, it is therefore possible to recover communications with computation, reducing the

overall communication cost. This mechanism is similar to task usage in OpenMP. Tasks can be delayed and the user can use synchronization (taskgroup/taskwait) to ensure their completion.

The first scenario (*IW*) is where an MPI process does an asynchronous receive (`MPI_Irecv`) and immediately waits (using `MPI_Wait`) for it to be satisfied. The second scenario (*ITCW*) is where an MPI process does an `MPI_Irecv` immediately followed by the execution of an OpenMP parallel region. One thread of the parallel region checks for the receive to be satisfied (using `MPI_Test`), while the others do some minor computation followed by a wait at the end. The third scenario (*ICW*) is the same as the first except `MPI_Irecv` is immediately followed by 500 µs computation before waiting. In this last case, we made sure that the overall computing had the same duration than in the second case – to allow direct comparison.

Figure 1 shows results from measuring the time spent in MPI_Wait in the three different MPI scenarios. The communication duration is the time from when `MPI_Irecv` is called to when `MPI_Wait` returns (or `MPI_Test` returns true). In our case, we focus on `MPI_Wait` time in order to measure the time needed to complete an MPI call relatively to the associated asynchronism construction. If we consider the scenarios run on a single node where MPI is using shared memory, it is clear that if `MPI_Test` is not called, the MPI runtime is less efficient for some reason. Not directly waiting is worse for small messages, due likely to



(a) OpenMPI 2.1.0 (SHM) (b) MPICH 3.3a2 (SHM)

(c) OpenMPI 1.8.8 (IB) (d) Intel MPI 17 (IB)

Fig. 1. Comparing our progress scenarios on the receiver side when running over both a shared memory segment and an Infiniband network (averaged 1000 times).

the extra 500 µs processing, and comparable for larger ones. Of course, these results are highly dependent on the underlying network. Maybe the results are an anomaly of running in MPI shared memory. However, if we repeated the experiments using InfiniBand in an dual-node configuration, the same pattern appears in Fig. 1(c).

MPI runtimes have to make a decision about how to implement waiting for asynchronous communication. The tradeoff has to do with how much overhead is spent in checking for communication completion, versus latency between when the communication actually completed and when it was detected by the MPI runtime. In other words, it is a decision about how to implement *progress* in the MPI runtime. What is seen in the graphs is the result of progress latency for the IW and ICW cases. In the case of ITCW, the `MPI_Test` acts like an immediate progress step. It should be able to take advantage of the overlap and that is what is observed.

The ramifications of these experiments is that progress is needed to achieve good performance in a heterogeneous computation context. More specifically, testing MPI requests is important for communication progress, but it pushes the responsibility for progress to the computational runtime (i.e., OpenMP), which must fill up the asynchronous periods as much as possible with work to get high performance.

How can we do this with OpenMP? Suppose we progressively insert MPI calls inside the parallel region, this while accounting for the requirement of progressing the MPI runtime. Our idea is to do this with our tasking patterns, iteratively increasing the functionality they offer. The extension of the OpenMP standard will then allow us to submit an increasingly complete DAG of execution and thus to prioritize more effectively the tasks carrying out MPI actions. For a working parallel example, we consider a 1D model (say a $10e6$ double array) evenly split between MPI ranks, where each rank has a core computation and ghost cells for communication to neighbor cells. In this case, each ghost cell might consists of 4096 doubles for each side with a periodic condition on the borders.

```
while(!finished && mpi_comm_complete != MPI_COMM_NUM){
 for( i = 0; i < MPI_COMM_NUM; i++ )
   if( ! Atomic_load_int( tab_flags[i] ) ){
     MPI_Test( &(tab_reqs[i]), &mpiflag, ↩
    MPI_STATUS_IGNORE);
     if(mpiflag && !Atomic_cas_int( &(tab_flag[i]), 0, 1 ↩
    )){
       Atomic_incr_int(mpi_comm_complete );
       compute_ghost_associated_part(i);
     }
   }
 finished = compute_core_part(); // yield
}
```

Listing 1.1. MPI AWARE Select (loop splitting)

Suppose we had to using OpenMP 2.0 and we wanted to mix MPI calls in an OpenMP parallel region. We could do something similar to what is illustrated in Listing 1.1. In this case, the loop computing the core computations would be separated. Then border communications would be progressed using `MPI_Test`, and associated border computation triggered on completion. Then if communications have not competed yet, the core calculation can be used to recover communications. In order to extend this MPI query polling in the `MPI_THREAD_MULTIPLE` case, we have based our selection on the basis of an atomic value table. The calculation phase ends when all the MPI communications and the associated actions are realized (i.e., computation of the border and MPI_Isend, but also the core part). The execution path is constrained according to MPI dependencies. However, two computing functions are effectively parallelized internally at the price of a critical section choosing the next action based on communication completion. This reduces the potential overhead of MPI communications by constraining OpenMP behavior. Indeed, to be able to improve granularity, the core compute function would have to be chunked, in order to regularly progress and check communication dependencies. This code is, in fact, doing different kinds of tasks, encouraging us to rely on OpenMP tasks.

```
1
#pragma omp parallel
3 {
  #pragma omp for nowait
5    for (i = 0; i < CORE_PART_NUM; i++)
       #pragma omp task
7      compute_core_part(i);

9  #pragma omp single
   {
11   while(mpi_comm_complete != MPI_COMM_NUM)
     {
13     for( i = 0; i < MPI_COMM_NUM; i++ )
         if( ! Atomic_load_int( tab_flags[i] ) )
15         #pragma omp task
           __progress_mpi_comm( i );
17       #pragma omp taskyield
     }
19 }
}
```

Listing 1.2. MPI AWARE Select (standard task)

Now, consider the use of OpenMP v3. Listing 1.2 shows multiple OpenMP tasks being created to handle a certain number of computing cores and multiple OpenMP tasks are dedicated to the progress of MPI communications. Moreover, thanks to the `taskyield`, MPI-related tasks are at most the number of MPI requests not completed. Dedicating a actual hardware core communications progression does not necessarily induce a penalty for the user code, especially when considering architectures with a large number of cores such as the Intel KNL

with 68 cores (272 hyper-threads). A hyper-thread, corresponds to 0.4% of a KNL – a totally acceptable overhead.

As we can not modify the task scheduler, MPI progress will not be multi-threaded or prioritized. In most OpenMP implementations, an OpenMP thread performs its own tasks before stealing from other threads. In our scenario, stealing of communication tasks will only occur when a thread will have completed its own tasks – actually yielding the desired behavior. In this configuration without priority, only the stealing mechanism can give us a form of priority. For instance, when running this code, the GOMP runtime did not allow the `taskyield` construct. As far as Intel OpenMP is concerned, it was not providing expected performance gains. When waiting for communication we expected to schedule computing-related tasks. These runtime limitations required us to explore another task approach presented below in order to correctly progress communications.

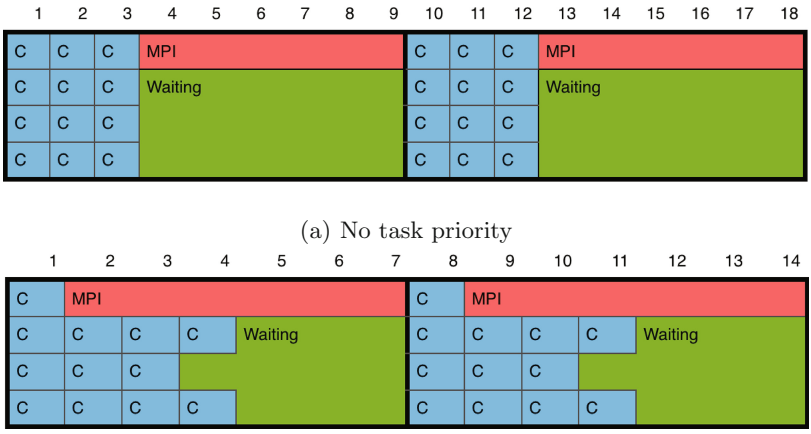
```
#pragma omp for nowait
for (i = 0; i < CORE_PART_NUM; i++)
  #pragma omp task priority(1)
   compute_core_part(i);

#pragma omp single nowait
{
   for( i = 0; i < MPI_COMM_NUM; i++ ){
#pragma omp task depend(inout: req_mpi ...) priority(100)
     {
       while( __mpi_request_not_match() )
       #pragma omp taskyield
     }
     if( i > MPI_COMM_SEND_NUM ){ // RECV REQUEST
#pragma omp task depend(inout:req_mpi ...) priority(100)
       {
          __compute_associated_border( i )
       }
#pragma omp task depend(inout: req_mpi ...) priority(100)
       {
          __send_ghost_associate( i );
       }
     }
   }
}
```

Listing 1.3. MPI AWARE Select (standard task)

Our initial idea was to rely on priorities and dependencies to pre-post MPI actions. To do so, valid and computable dependencies are required at compilation time. This leads to a problem when considering communications, a given MPI process may have a varying neighboring (mesh corners) while these dependencies have to be known at compilation time (no dynamic dependencies). In our example, `MPI_Requests` are static variables. Aware of `taskyield` limitations,

we proposed in Listing 1.3 with OpenMP 4.x in mind. This is a version based on `single`, allowing us to force a thread to poll MPI Request. We can use the `taskgroup` ensures that all threads participate in the execution of the sets of tasks, including the one testing for MPI communications. Eventually, the send task has two dependencies, ensuring that the previous send is complete before issuing the next.



(a) No task priority

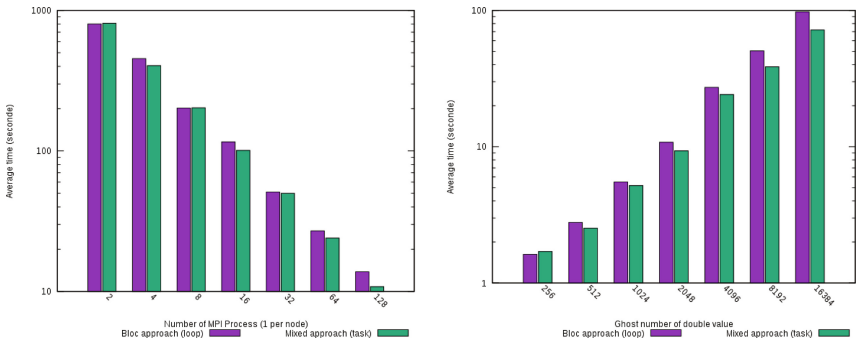
(b) With high priority on MPI task

Fig. 2. Interest of task priority with heterogeneous task

If we consider an abstract time unit with a computing task that is worth 1 unit and an MPI task worth 6, then looking at Fig. 2 observe that the choice of scheduling can have an impact on the total execution time. We have illustrated the execution time of four threads with 12 computational tasks and 1 MPI task per time step. It is recognized that a greedy algorithm favoring the task taking the most time generally allows to reach a relevant local minimum. The developer can not make assumption about the behavior of the OpenMP support. For this reason, OpenMP priorities are of interest to handle such heterogeneous tasks.

4 Evaluation

With the introduction of OpenMP tasks, it begs the question of how tasks would compete with the traditional parallel loop approach. In one respect, by avoiding successive fork-join, tasks are able to improve the overall scheduling. Returning to our reference benchmark, in a loop-based version, Isend/Irecv are posted, the core part is computed, communications waited on, and then borders processed. In the task-based version, the tasks are pushed immediately when the progress thread completes a test. Thus, only a single parallel region with a computation split in tasks is required.



(a) Over MPI Processes (b) Over Ghost size (64 Processes)

Fig. 3. Comparison of our bulk-synchronous (loop parallelism) and our proposed task-based approach over both process count and message size (fixed at 64 MPI processes).

We compared these two versions on an Intel Sandy-bridge machine up to 2048 cores. Each dual-socket node gathers 16 cores on which we ran 16 OpenMP threads. In order to generate the results presented in Fig. 3, we ran the code for 1000 timesteps, conducted 10 experiments, and averaged the execution times. We observe that in this first case the task approach is better than the loop one when the number of nodes is higher than 16, this despite one core is used to progress communications. We believe this performance difference is due to the increasing noise in MPI messages, creating irregularities in the communication scheme. Moreover, as the number of cores increases, the overall computation decreases (strong scaling), due likely to increase communication jitter.

To get a sense of effective MPI overlap, we increased the ghost cell size to increase the size of communication. We observed that MPI overlaps is almost null with the runtime that we used on the target machine (OpenMPI 1.8.8), justifying our efforts to integrate progress inside our parallel OpenMP constructs. Figure 3(b) shows the effects as we see performance gains with greater communications sizes, demonstrating the importance of progressing MPI messages.

5 Progress and OpenMP

MPI communication progress is a key factor in hybrid parallelism. Consequently, in order to take advantage of asynchronous messages within an MPI+OpenMP program, communications must be explicitly progressed through MPI runtime calls (`MPI_Test`, `MPI_Probe`). Not doing so shifts most of the message completion responsibility to the actual `MPI_Wait` operation (at least in the configurations we measured). This can all but eliminate any benefits in overlapped communication and communication. Our proposed remedy to overcoming this problem is to utilize task-based constructions. However, additional constructs in OpenMP may help solving this progress issue and more generally support better runtime stacking.

```
void idle( void *prequest ) {
  if(__mpi_request_match(prequest)){
    omp_trigger("ghost_done");
    return 0;
  }
  return 1;
}

#pragma omp parallel progress(idle, &request)
{
   #pragma omp noprogess
   {
      MPI_Wait(request, MPI_STATUS_IGNORE);
      omp_trigger("ghost_done");
   }
   #pragma omp task depends(inout:"ghost_done")
   {/* BORDER */}
   #pragma omp task
   {/* CENTER */}
}
```

Listing 1.4. Proposed implementation for a progress enabled OpenMP

In general, as presented in Listing 1.4, OpenMP could gain from a notion of progress. Indeed, one could define what processing has to be done to satisfy task dependencies, letting the runtime invoke the `progress` function to trigger dependencies. In order to realize this idea, two things are needed. First, a progress parameter would be included for parallel regions to define which function should be called when the runtime is idle or switching between tasks. This should be a function as it contains code which may not be executed if not compiled with OpenMP support; if this function returns "0" it is not called further, if it is "1" it continues to be called as there is work remaining. In this case, the otherwise ignored `noprogress` code section is executed, replacing the non-blocking progress calls with blocking ones.

Second, we need *named dependencies* between tasks. This is because we want another runtime to satisfy a dependency which cannot be known at compilation time as an address, for example, *"ghost_done"*. To do so, we define `omp_trigger` which satisfies a named dependency. Using this simple construct, we are then able to express in a compact manner, a communication dependency with a direct fallback to a blocking version if OpenMP is not present. This abstraction seems reasonable based on our experiments, and we are in the process of implementing this feature to validate it further.

6 Related Work

Scalable, heterogeneous architectures are putting increasing pressure on the pure MPI model [5]. Hybrid parallel programming is necessary to expose multiple

levels of parallelism, inside and between nodes. However, in order to leverage and mix our existing models, their runtime systems must interoperate more efficiently.

Hybridization appeared with accelerators and programming languages such as Cuda or OpenCL [18]. Here the host node retains its own cores and memory for program execution, but off-loads computation and data to the accelerator device. Accelerators such as GPUs are generally energy efficient and expose a high level of stream oriented parallelism. One issue with their use is the need for explicit transfers to and from the device, requiring important programming efforts to manage data. Moreover, CPU resources might be underutilized if only used to move data. This argues for another hybrid parallelism level. Pragma-based models such as OpenMP target [3,4], OpenACC [20], Xkaapi [11] and StarPU [2] proposed abstractions combining GPUs and CPUs in an efficient manner, abstracting data-movements.

For the most part, MPI has not been integrated in shared-memory computations. Rather, an MPI process is primarily seen as a container for the shared computation, and most programs evolved from MPI to handle new parallelism models [6]. For these practical reasons, there were fewer efforts to embed MPI in another model (e.g., X+MPI), versus expressing parallelism inside MPI processes (MPI+X). The advent of many-core architectures, such as the Intel MIC [9] and the Intel KNL enforced the use of larger shared-memory contexts, requiring some form of hybridization. MPI+OPenMP is nowadays accounting for a large number of applications, nonetheless, neither MPI or OpenMP have collaborative behaviors. Both of them are distinct runtimes with their respective ABI/API. However, there are several programming models aimed at providing an unified view of heterogeneous or distributed architectures: Coarray Fortran [17], Charm++ [13], HPF [15], Chapel [7], Fortress [1] and X10 [8]. Several of them rage various communication models, including message passing (MPI) and (partitioned) global address space ((P)GAS).

A complementary approach is based on Domain Specific Languages [10] which is aimed at abstracting parallelism expression [12] in order to "free" codes from programming model constraints, for example by targeting multiple models [19]. There is a wide range of such specialized languages with clear advantages, however, they transpose the dependency from a model to a dedicated language with its own constraints [14].

Our work is close to the idea developed by Marjanovic et al. [16], they proposed a set of pragma to improve the processing of non-blocking MPI communications in a multithreaded context. The use of new pragmas requires a specific compiler and results in a loss of portability. Our initial solutions based on tasks differs in the sense that they are only based on the use of standard OpenMP pragmas without any hypothesis of specific executive support mechanisms.

Model mixing and unified models, in general, is a very active research area with a wide variety of approaches. In this paper, we focused on two common building blocks, MPI and OpenMP, trying to see how MPI could be embedded

inside OpenMP constructs in an efficient manner. In a way, this takes a reverse approach when most efforts tend to embed OpenMP inside MPI.

7 Conclusion

In this paper, we first introduced the need for hybridization in parallel applications. Indeed, when scaling multiple nodes gathering hundreds of cores, the MPI+X paradigm becomes compulsory to limit both memory and communication overhead. Unfortunately, most MPI+OpenMP codes rely on alternating phases between communication and compute. This can constrain parallel performance due to the fork-join nature of OpenMPI parallelism and the sequentiality of MPI phases outside of a parallel region. Instead, we explored an alternative approach relying on tasks and how they could help to maintain MPI progress during OpenMP parallel computation. With the latest OpenMP version, multiple approaches are possible to mix MPI with OpenMP.

Our hybridization ideas essentially advocate that the program become a Directed Acyclic Graph (DAG) to be scheduled by the OpenMP runtime. The DAG is made of tasks that combine processing from both OpenMP (computation) and MPI (communication). However, such combination is not natural in OpenMP, particularly when considering `MPI_Request` handles which are generated dynamically during the execution. Indeed, OpenMP does not allow tasks dependencies to be expressed on the fly, instead, they have to be resolvable at compilation time. Consequently, in this paper, we propose three different approaches to mixing OpenMP tasks and MPI despite this static dependency resolution. Each approach is described and evaluated with a simple benchmark. The measurements show that a task-based approach was beneficial to the overall execution, in particular, by allowing greater computation overlap. However, a key issue is MPI progress. Efficient hybrid execution (MPI+OpenMP) can only be achieved if MPI calls are regularly invoked during parallel OpenMP computation, as our task-based examples demonstrate.

8 Future Work

In general, effective runtime inter-operation and stacking for hybrid parallel programming requires interactions to coordinate progress. It is important to consider then the integration of this support in programming standards. For instance, in our study of MPI+OpenMP, if the `taskyield` call could be defined as an arbitrary function, it would be possible for the OpenMP runtime to notify the MPI runtime that it may progress communications. With this progress issue solved, dynamic (on request addresses) or label-based dependencies would be an alternative to the jump-table we relied on in this paper. There can be side effects when combining two runtimes that need some additional support to resolve.

References

1. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.W., Ryu, S., Steele, G.L., Tobin-Hochstadt, S.: The Fortress language specification. Tech. report, Sun Microsystems, Inc., version 1.0, March 2008
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03869-3_80
3. Ayguade, E., et al.: A proposal to extend the OpenMP tasking model for heterogeneous architectures. In: Müller, M.S., Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 154–167. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02303-3_13
4. Bertolli, C., Antao, S.F., Eichenberger, A.E., O'Brien, K., Sura, Z., Jacob, A.C., Chen, T., Sallenave, O.: Coordinating GPU threads for OpenMP 4.0 in LLVM. In: Proceedings of the 2014 LLVM Compiler Infrastructure in HPC, LLVM-HPC 2014, pp. 12–21. IEEE Press, Piscataway (2014). http://dx.doi.org/10.1109/LLVM-HPC.2014.10
5. Besnard, J.B., Malony, A., Shende, S., Pérache, M., Carribault, P., Jaeger, J.: An MPI halo-cell implementation for zero-copy abstraction. In: Proceedings of the 22nd European MPI Users' Group Meeting, EuroMPI 2015, NY, USA, pp. 3:1–3:9 (2015). http://doi.acm.org/10.1145/2802658.2802669
6. Brunst, H., Mohr, B.: Performance analysis of large-scale OpenMP and hybrid MPI/OpenMP applications with Vampir NG. In: Mueller, M.S., Chapman, B.M., Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005. LNCS, vol. 4315, pp. 5–14. Springer, Heidelberg (2008). doi:10.1007/978-3-540-68555-5_1
7. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. Int. J. High Perform. Comput. Appl. **21**(3), 291–312 (2007). http://dx.doi.org/10.1177/1094342007078442
8. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. **40**(10), 519–538 (2015). http://doi.acm.org/10.1145/1103845.1094852
9. Duran, A., Klemm, M.: The intel many integrated core architecture. In: 2012 International Conference on High Performance Computing Simulation (HPCS), pp. 365–366, July 2012
10. Fowler, M.: Domain-Specific Languages. Pearson Education, Boston (2010)
11. Gautier, T., Lima, J.V.F., Maillard, N., Raffin, B.: XKaapi: a runtime system for data-flow task programming on heterogeneous architectures. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, pp. 1299–1308, May 2013
12. Hamidouche, K., Falcou, J., Etiemble, D.: Hybrid bulk synchronous parallelism library for clustered SMP architectures. In: Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications, HLPP 2010, NY, USA, pp. 55–62 (2010). http://doi.acm.org/10.1145/1863482.1863494
13. Kale, L.V., Krishnan, S.: Charm++: a portable concurrent object oriented system based on c++. SIGPLAN Not. **28**(10), 91–108 (1993). http://doi.acm.org/10.1145/167962.165874

14. Karlin, I., Bhatele, A., Keasler, J., Chamberlain, B.L., Cohen, J., Devito, Z., Haque, R., Laney, D., Luke, E., Wang, F., Richards, D., Schulz, M., Still, C.H.: Exploring traditional and emerging parallel programming models using a proxy application. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, pp. 919–932, May 2013
15. Loveman, D.B.: High performance Fortran. IEEE Parallel Distrib. Technol. Syst. Appl. **1**(1), 25–42 (1993)
16. Marjanović, V., Labarta, J., Ayguadé, E., Valero, M.: Overlapping communication and computation by using a hybrid MPI/SMPSS approach. In: Proceedings of the 24th ACM International Conference on Supercomputing, ICS 2010, NY, USA, pp. 5–16 (2010). http://doi.acm.org/10.1145/1810085.1810091
17. Numrich, R.W., Reid, J.: Co-array Fortran for parallel programming. SIGPLAN Fortran Forum **17**(2), 1–31 (1998). http://doi.acm.org/10.1145/289918.289920
18. Stone, J.E., Gohara, D., Shi, G.: OpenCL: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(3), 66–73 (2010)
19. Sujeeth, A.K., et al.: Composition and reuse with compiled domain-specific languages. In: Castagna, G. (ed.) ECOOP 2013. LNCS, vol. 7920, pp. 52–78. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39038-8_3
20. Wienke, S., Springer, P., Terboven, C., Mey, D.: OpenACC — first experiences with real-world applications. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 859–870. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32820-6_85