Communication-Aware Task Scheduling Strategy in Hybrid MPI+OpenMP Applications

Romain Pereira[1,3(✉)], Adrien Roussel[1,2], Patrick Carribault[1,2], and Thierry Gautier[3]

[1] CEA, DAM, DIF, 91297 Arpajon, France
`{romain.pereira,adrien.roussel,patrick.carribault}@cea.fr`
[2] Université Paris-Saclay, CEA, Laboratoire en Informatique Haute Performance pour le Calcul et la simulation, 91680 Bruyères-le-Châtel, France
[3] Project Team AVALON INRIA, LIP, ENS-Lyon, Lyon, France
`thierry.gautier@inrialpes.fr`

Abstract. While task-based programming, such as OpenMP, is a promising solution to exploit large HPC compute nodes, it has to be mixed with data communications like MPI. However, performance or even more thread progression may depend on the underlying runtime implementations. In this paper, we focus on enhancing the application performance when an OpenMP task blocks inside MPI communications. This technique requires no additional effort on the application developers. It relies on an online task re-ordering strategy that aims at running first tasks that are sending data to other processes. We evaluate our approach on a Cholesky factorization and show that we gain around 19% of execution time on an Intel Skylake compute nodes machine - each node having two 24-core processors.

Keywords: MPI+OpenMP · Task · Scheduling · Asynchronism

1 Introduction

High Performance Computing (HPC) applications target distributed machines, which inevitably involve inter-node data exchanges that can be handled by MPI (Message Passing Interface). But, at compute-node level, the number of cores is increasing, and task programming models seem to be well-suited for efficient use of all computing resources and to satisfy the needs of asynchronism. Since 2008, OpenMP [1,14] defines a standard for task programming. This leads to codes that finely nest MPI communications within such OpenMP tasks. Furthermore, OpenMP 4.0 introduced data dependencies between tasks. Thus, within parallel regions exploiting tasks, applications can be seen as a single global data-flow graph which is distributed across MPI processes, where each process has its own OpenMP task scheduler with no view of the global graph. This may result in poor performance [12] and even deadlocks [19].
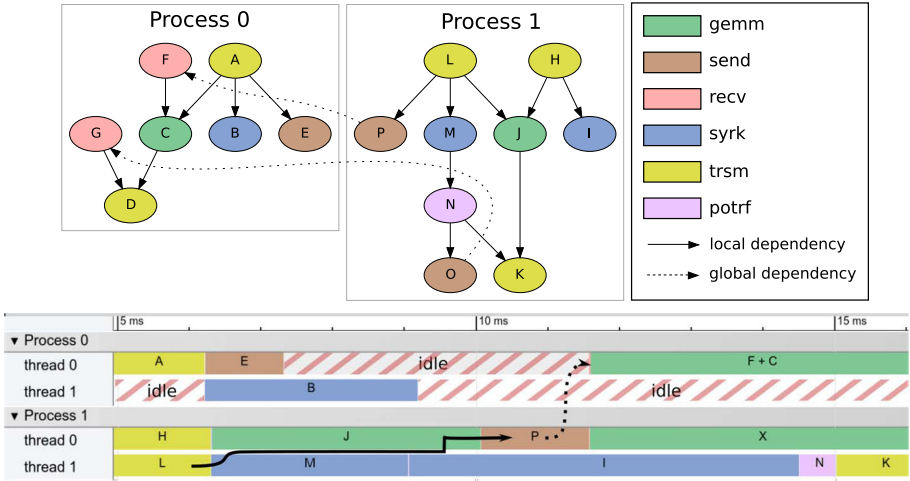
Fig. 1. Top: sub-graph of a distributed blocked Cholesky factorization mapped onto 2 MPI Ranks (matrix size: 2048×2048 with tile of size 512). Bottom: Gantt chart with 2 threads per MPI rank.
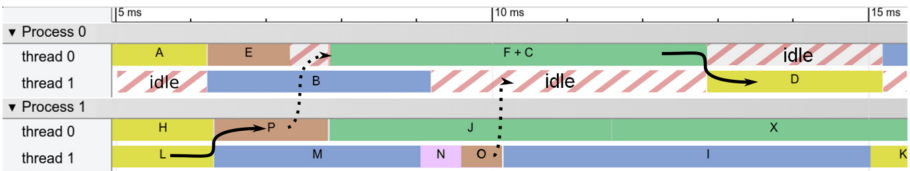

Fig. 2. Alternative scheduling for the graph in Fig. 1.

Deadlocks can be due to the loss of cores when threads execute blocking MPI calls within OpenMP tasks [11]. Several solutions address this issue [16,18,20] and enable working MPI+OpenMP(tasks) codes, but performance issues remain.

Task scheduling in this hybrid context can significantly improve the overall performance. As an example, Fig. 1 presents a subgraph of the task dependency graph (TDG) for a Cholesky factorization [19], and its scheduling trace on 2 processors of 2 threads each. Tasks are scheduled following a standard First In, First Out (FIFO) policy: once precedence constraints are resolved, the first tasks created are scheduled first. This policy leads to 61% idle time on `Process 0`, partially because it is waiting for data from `Process 1`. A similar result would be obtained with regular OpenMP runtimes (GNU-OpenMP [6], LLVM-OpenMP [8]) which uses a mix of FIFO and Last In, First Out (LIFO) policies. It is possible to reduce this idle time to 36% by adapting the scheduling policy to prioritize inter-process edges (see Fig. 2).

This paper proposes a communication-aware task re-ordering strategy for OpenMP that aims at reducing idle periods in hybrid MPI+OpenMP (tasks)

applications by favoring tasks on a path to communications. This strategy relies on hybrid scheduling techniques and proposes an automatic TDG prioritization based on communication information. Our solution heavily leverages runtime interoperations but requires no OpenMP/MPI extensions and no further efforts on the user side on tasks prioritization. Section 2 presents related work. Then, Sect. 3 highlights our task scheduling strategy and Sect. 4 exposes its implementation and evaluation. Finally, Sect. 5 concludes and discusses future work.

2   Related Work

The Dominant Sequence Clustering (DSC) [22] was proposed as a heuristic for scheduling tasks on an unlimited number of distributed cores. This algorithm distributes a fully-discovered TDG onto cores - the clustering phase - and prioritizes tasks using global bottom and top levels. Our paper focuses on hybrid MPI+OpenMP(tasks) which are the widely used standards in the HPC community. With this programming model, the clustering is done by the user which distributes OpenMP tasks onto MPI processes. Each OpenMP tasks scheduler only has a view on its local subgraph (*i.e.* its cluster). The tasks *global* bottom and top levels (*blevel*, *tlevel*) are not known, which leads us to prioritize the TDG using purely local information.

MPI+OpenMP(tasks) model may lead to a loss of thread, when a thread executes blocking MPI code within an OpenMP task [11,12]. Many works addressed this issue [4,9,11,16,18,19]. Some approaches [4,11] consist of marking communication tasks from user codes and dedicating threads to communication or computation. This guarantees that both communications and computations progress and fix deadlock issues due to the loss of threads. However, it requires user-code adaptations and creates load balancing issues between communication and computation threads. In 2015, MPI+ULT (User Level Thread, Argobots) [9] proposed to run MPI code within a user-level thread, and make it yield whenever an MPI communication blocks. In OpenMP, yielding can be achieved using the `taskyield` directive. Schuchart *et al.* [19] explored various implementations of it, and having implementations that effectively suspend, enables the expression of fine MPI data movement within OpenMP tasks. This resulted in a more efficient implementation of the blocked Cholesky factorization with fewer synchronizations and led to new approaches on MPI+OpenMP(tasks) interoperability, such as TAMPI [18] and MPI_Detach [16]. TAMPI was proposed as a user library to enable blocking-tasks pause and resume mechanism. It transforms calls to MPI blocking operations to non-blocking ones through the `PMPI` interface and interoperates with the underlying tasking runtime - typically using the `taskyield` in OpenMP, or `nanos6_block_current_task` in Nanos6. The authors of [16] proposed another interoperability approach using the `detach` clause, which implies MPI specifications extensions to add asynchronous callbacks on communications completion, and also user code adaptations.

Among all these works, our solution on the loss of threads issue differs from [4, 11,16]: we aim at no user code modifications, and to progress both communications

and computations by any thread opportunistically. Our approach is more likely a mix of [9,16,18] with automation through runtime interoperations. This part of our strategy is detailed in Sect. 3.1.

Other reasons can lead to threads idling. For instance, an unbalanced distribution of work between nodes leads to threads idling. CHAMELEON [7] is a reactive task load balancing for MPI+OpenMP tasks applications and enables OpenMP tasks migration between MPI processes. Another idling reason could be communication synchronization. A dynamic broadcast algorithm for task-based runtimes was proposed in [5] which aims at no synchronizations. Their algorithm consists of aggregating data-send operations to a single request, which holds all the recipients' information. Our work assumes that the task load is balanced across MPI processes and that applications only use asynchronous point to point communications. Thus, our scheduling strategy should be used alongside [5,7] to achieve the best performances in real-life applications.

Asynchronous communications progression in hybrid MPI+OpenMP tasks programming is discussed by David Buettner et al. in [2]. They proposed an OpenMP extension to mark tasks that contain MPI communications. This allows them to asynchronously progress MPI communication on every OpenMP scheduling point. We retrieved this technique in our paper, as part of our execution model. However, we propose it without the need to mark tasks on the user side, by adding runtime interoperability.

3 Task Scheduling Strategy

We target applications that nest MPI point-to-point communications within OpenMP dependent tasks. Our strategy aims at scheduling first tasks that send data to reduce idle time on the receiving side. For this purpose, let us denote respectively `recv-tasks` and `send-tasks` tasks that contain `MPI_Recv` and `MPI_Send` calls, or their non-blocking version - `MPI_Irecv` and `MPI_Isend` - paired with a `MPI_Wait`. We will discuss in Sect. 3.3 how to identify them. This section starts by the presentation of the assumed interoperation between OpenMP and MPI. Then it exposes, in a progressive way, different policies to adapt task scheduling in order to send data at the earliest: through manual (using OpenMP `priority` clause) or automatic computation of the required tasks annotation.

3.1 Interoperation Between MPI and OpenMP Runtimes

Each MPI process has its own OpenMP scheduler, which executes tasks according to their precedence constraints (expressed through the `depend` clause) and their priorities. To address the loss of cores issue introduced by a blocking MPI communication calls, and to keep asynchronous communication progression, we propose a mix of User Level Threads (ULT), TAMPI, and MPI_Detach [2,9,16,18]. On the MPI runtime side, whenever a thread is about to block, it injects a communication progression polling function inside OpenMP, to be

called on every scheduling point - as it was proposed by D. Buettner and al. [2]. Moreover, the MPI runtime notifies the OpenMP runtime that the current task must be suspended. The MPI communication progresses on each OpenMP scheduling point, and whenever it completes, MPI notifies OpenMP runtime that the suspended task is eligible to run again. This solution enables the progression of both communication and computation asynchronously, by any threads opportunistically.

3.2   Manual Policies

This section presents a preliminary strategy to favor communication tasks based on source-code modification to evaluate the overall approach. To express the fact that a specific task contains an MPI communication or it is on a path to such operation, this strategy relies on the OpenMP `priority` clause. Thus, this section introduces 3 manual policies to annotate programs in order to fix the priorities of each task. We call this process *TDG prioritization*: Fig. 3 resumes the priorities setting by the policies.
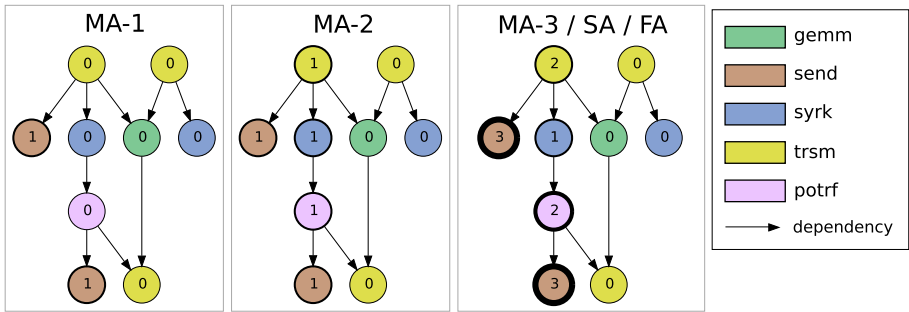

Fig. 3. Task priorities set by the various policies for the graph Fig. 1 - the value on the nodes corresponds to the task priority

MA-1. Relies on a binary priority `0` (the lowest) and `1` (the highest). Tasks with priority `1` are always scheduled - once dependencies are fulfilled - over the ones with priority `0`. Here, the user has to manually sets priority `1` on `send-tasks`, and sets no priority on other tasks resulting in a `0` low priority internally. This way, whenever the scheduler has multiple tasks marked as ready (*i.e.,* with fulfilled dependencies), it may schedule `send-tasks` if there are any. This manual prioritization policy is presented in [17].

MA-2. In similar situations to the one depicted in Fig. 1, we would also like to prioritize all the tasks that precede a `send-task`, to fulfill its dependencies to the earliest. In this specific case, the user should prioritize the path (L, M, N)

to resolve `O` dependency constraint. MA-2 consists in setting to `1` the priority on `send-tasks` and on all their predecessors recursively.

MA-3. MA-2 does not allow to finely distinguish `send-tasks` and their path. However, this is important since sent data may be needed by a remote node earlier or later. The earlier the data is needed in a remote rank, the earlier it should be sent. MA-3 relies on discrete priorities to prioritize various `send-tasks` distinctly. One way is to use their depth in the local TDG: the shorter the path to the `send-task` in the local TDG, the higher the priorities on its paths. This prioritization follows Algorithm 1 and is illustrated in Fig. 3. `Send-tasks` are set with the maximum priority (line 3). If the task has no successor (lines 4–5) or if no path from it leads to a send-task (lines 8–10), then we set no priority for it. Otherwise, there is a path from the current task to a `send-task` and we set its priority by decrementing the value of the highest priority among its successors (line 12).

In practice, the MA-3 strategy requires the user to annotate every task on a path that contains a `send-task`.

Algorithm 1. Task prioritization

Input: Task T
Output: P(T) - the priority of T
 1: **function** COMPUTEPRIORITY(T)
 2:     **if** T is a send-task **then**
 3:         | **return** omp_get_max_priority()
 4:     **if** $Successors(T) = \emptyset$ **then**
 5:         | **return** 0
 6:     **for all** $S \in Successors(T)$ **do**
 7:         | P(S) = ComputePriority(S)
 8:     M = max($\{P(S) \mid S \in Successors(T)\}$)
 9:     **if** $M = 0$ **then**
10:         | **return** 0
11:     **else**
12:         | **return** M - 1

3.3 (Semi-)Automatic Policies

While MA-3 reduces the idling periods by sending data to the earliest, manually prioritizing the TDG is tedious to implement at user level. Users will have to manually compute the depth of each tasks in the local TDG, and setting the `priority` clause accordingly. From the runtime point of view, this information could be tracked. So, we propose two runtime automations on the TDG prioritization to reduce user programming efforts to identify predecessors of a task as well as to identify communication tasks.

SA. In the Semi-Automatic (SA) approach, the user simply marks `send-tasks` with an arbitrary priority. Once a task construct is encountered, the runtime is guaranteed that all of its predecessors were already created too. So, the runtime internally sets its priority to the highest value, and automatically propagates it through the TDG following Algorithm 1.

FA. SA enables a more straightforward MA-3 implementation but it still requires the user to mark `send-tasks`. The Fully-Automatic (FA) approach enables a TDG prioritization similar to MA-3, but with absolutely no hints given by the users to the runtime, by adding fine collaboration between MPI and OpenMP runtimes. At execution time, whenever MPI is about to perform a send operation, it notifies the OpenMP runtime. If the current thread was executing an explicit task, it registers its profile with information such as its:

– size (`shared` variables)
– properties (tiedness, final-clause, undeferability, mergeability, if-clause)
– parent task identifier (the task that spawns current task)
– number of predecessors (fully-known at run-time)
– number of successors (may be incomplete)

Then, future tasks may be matched with registered profiles to detect `send-tasks`. This approach uses full-matching on the size, the properties, the parent task identifier, and the number of predecessors. It mainly targets iteration-based applications, where `send-tasks` profiles are likely to be identical between iterations.

The `send-tasks` cannot be detected until a task with a similar profile was scheduled, performed an MPI send operation, and registered its profiles in the OpenMP runtime. So unlike the `SA` policy, the prioritization cannot be done on task constructs with `FA`. We propose to perform it asynchronously during idle periods. This way, the runtime is more-likely to have detected `send-tasks` when performing the matching, and idle periods are overlapped by the prioritization without slowing down ready computations. Algorithm 2 is a single-threaded asynchronous TDG prioritization proposal. The parameter `ROOTS` corresponds to the task-nodes from which the prioritization should start, *i.e.* the blocking tasks. Line 9 to 17 consists of breadth-first-searching leaves, so we have them sorted by their depth in the TDG. Line 18 to 24 goes up from founded leaves to roots, matching tasks with registered profiles and propagating the priority to predecessors.

Table 1. Summary of approaches

Policies	MA-1	MA-2	MA-3	SA	FA
send-tasks	u	u	u	u	r
send-tasks path	N/A	u	u	r	r

u - user/manual
r - runtime/automatic

Algorithm 2. Priority propagation (single-thread, during idle periods)

1: **Variables**
2:     List D, U   ▷ D, U stands for DOWN, UP
3:     Task T, S, P
4:
5: **procedure** PRIORITIZE(ROOTS)   ▷ Prioritize the TDG from given root nodes
6:     D = [], U = []   ▷ Empty lists
7:     **for** T in ROOTS **do**
8:       Append T to D   ▷ Add T to the tail of D
9:     **while** D is not empty **do**
10:       T = D.pop()   ▷ Pop T from D's head
11:       **if** T has successors **then**
12:         **for** S in $Successors(T)$ **do**
13:           **if** S is not VISITED **then**
14:             Mark S as VISITED
15:             Append S to D
16:       **else**
17:         Append T to U
18:       **while** U is not empty **do**
19:         **if** T is not queued **then**
20:           Set T.priority   ▷ match with registered task profiles
21:           **for** P in $Predecessors(T)$ **do**
22:             **if** P.priority < T.priority - 1 **then**
23:               P.priority = T.priority - 1
24:               Prepend T to U

3.4 Summary

Table 1 summarizes the different approaches of computing priorities on tasks that perform MPI communications and tasks that contribute to execute communications (through OpenMP task dependencies). Almost all policies require user modifications of the application source code (through `priority` OpenMP clause) to mark tasks that perform send operations except the **FA** strategy that automatically detects such tasks by comparing profiles with previously-executed tasks. Furthermore, marking the whole path from the source task to the ones that perform MPI operations can be done manually (approaches **MA2** or **MA3**) or automatically (approaches **SA** or **FA**).

4 Implementation and Evaluation

4.1 Implementation

The scheduling strategy and the different policies presented in Sect. 3 were implemented into MPC [15]: a unified runtime for MPI and OpenMP [1]. It is based on hierarchical work-stealing, similar to [13,21], where red-black tree

---

priority queues are placed at the different topological levels built from the hardware topology [10]. Each thread is assigned to multiple queues, and steal tasks from other queues when it falls idle. The task with the highest priority is popped from the selected queue.

To implement the interoperability approach presented Sect. 3.1, we made two modifications to MPC framework. First of all, we added an MPC-specific OpenMP entry-point to suspend the current task until an associated `event` is fulfilled - `mpc_omp_task_block(omp_event_t event)`. When a thread is about to block on an MPI call, it suspends its current task through this routine. The communication progresses, and on completion, `omp_fulfill_event(omp_event_t event)` is called so that the associated task is eligible to resume.

Furthermore, to avoid deadlocks, we added contexts to OpenMP tasks in MPC based on a modified version of the `<ucontext>` C library to handle MPC-specific TLS [3]. This add some extra instructions on tasks management measured using `Callgrind`. For each task, contexts added ~ 200 *instructions* on launch, ~ 2000 *instructions* on the first suspension, and ~ 200 *instructions* each time a task suspends.

Based on these modifications, we implemented the `MA-1`, `MA-2`, `SA` and `FA` approaches presented in the previous section. However, `MA-3` was not implemented since it is too tedious on the user-side.

Task prioritization is done synchronously on task construct in the `SA` policy. In the `FA` policy it is done asynchronously: whenever a thread falls idle, it runs Algorithm 2 with `ROOTS` being the list of tasks suspended through `mpc_omp_task_block`.

4.2   Evaluation Environment

We present the result of multiple experiments on the fine-grained blocked Cholesky factorization benchmark. We denote n the size of the matrix to factorize, and b the size of the blocks. The time complexity of the factorization is $O(n^3)$, and the memory used is about $8n^2$ bytes.

All experiments run onto Intel Skylake nodes (two 24-core Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70 GHz, with 96 GB of DDR). Interconnection network is a Mellanox ConnectX-4 (EDR 100Gb/s InfiniBand) system. MPC (commit 702ce5c2) was configured with optimizations enabled, and compiled with GCC 7.3.0. We forked J. Schuchart fine-grained blocked Cholesky factorization benchmark.[2] The benchmark is compiled with MPC patched GCC [3], linked with Intel Math Kernel Library (MKL 17.0.6.256), MPC-MPI, and MPC-OpenMP. Each run uses SLURM `exclusive` parameter, which ensures that no other jobs may run concurrently on the same node. Each time corresponds to medians taken on 20 measurements.

To evaluate our strategy and different policies of Sect. 3 we compare measured performance against the FIFO reference policy, previously explained in Sect. 1.

This policy does nothing except that the MPC MPI and OpenMP runtimes interoperate to avoid deadlocks.
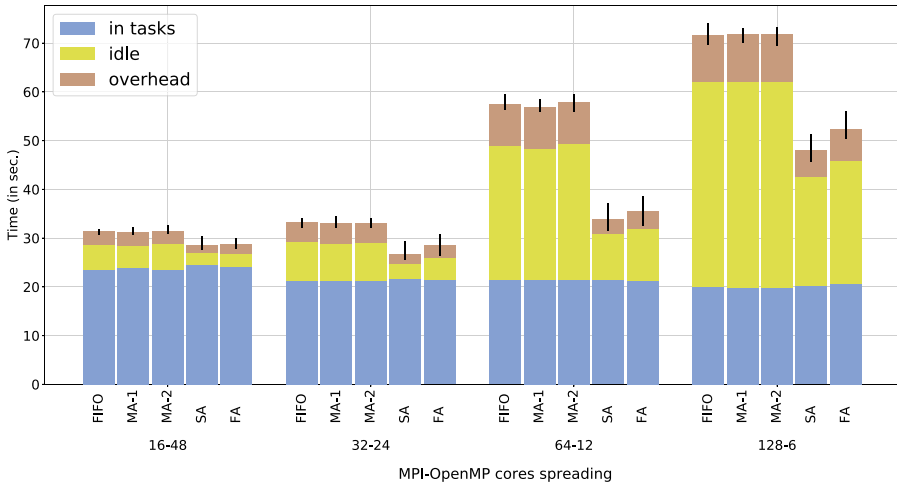

Fig. 4. Cholesky factorisation time based on the prioritization policy, and the MPI/OMP cores spreading on 16 Skylakes nodes (*e.g.*, 16–48 stands for 16 MPI ranks of 48 threads each) - with a matrix of size n = 131072, and blocks of size 512.

4.3   Experimental Results

Figure 4 shows the impact of cores spreading between MPI and OpenMP on the performance of each prioritization policy described in Sect. 3. The time was measured using a tool that traces every MPC-OpenMP tasks events and replay the schedule post-mortem to extract `in-tasks` (time spent in tasks body - MKL computation, non-blocking MPI communications initialization), `idle` (time spent outside tasks with no ready-tasks - idling, communications progression, prioritizations in `FA`), and `overhead` categories (time spent outside tasks with ready-tasks - tasks management, communications progression). The matrix size is $n = 131072$, and 16 fully-allocated Skylake nodes. The benchmark ran with different spreading configurations. On the left-most bars, there are 16 MPI ranks of 48 OpenMP threads each (1 MPI rank per node). On the right-most bars, there are 128 MPI ranks of 6 threads each (8 MPI ranks per node). In configurations with multiple MPI ranks on the same node, OpenMP threads of the same MPI rank always are on the same NUMA node. Note that the amount of computation between each configuration remains constant, there are precisely 2.829.056 OpenMP computation tasks in each run distributed across MPI ranks. The number of communication tasks increases with the number of MPI ranks and is depicted in Table 2.

Table 2. Number of point-to-point communication tasks in Fig. 4 runs

Cores spreading (MPI-OpenMP)	16–48	32–24	64–12	128–6
P2P communication tasks (overall)	388.624	640.632	892.640	1.372.880

This result demonstrates that tasks prioritization in MPI+OpenMP tasks applications can have significant impact on performance (as predicted in Figs. 1 and 2). First of all, `MA-1` and `MA-2` policies are not sufficient and they do not improve performance over the baseline `FIFO` policy. By prioritizing `send-tasks` and their path, the policies `SA` and `FA` significantly reduce idle periods. For instance in the `32-24` spreading, the total execution time and the idle time respectively are 33.2s and 8.0s for the `FIFO` policy, 28.4s and 4.4s for `FA` policy. However, being fully automatic has some costs. The `FA` policy is never as good as the `SA` one, with up to a 8% overhead in the `128-6` spreading. For `FA`, the prioritization only occurs once some `send-tasks` execution, their profile is registered and eventually a thread becomes idle to set and propagate the task priority. It means there is no prioritization for the first executed tasks. Moreover, the profile registering and matching mecanisms induce some overhead.

The 32-24 scheme reaches the best performance thanks to NUMA domain data-locality. MPC-MPI also optimizes intra-node rank exchanges, processing them in shared memory.

Table 3. Execution times of runs in Fig. 5

Number of MPI ranks	1	2	4	8	16	32
FIFO	22.28 s	24.05 s	25.82 s	26.46 s	31.02 s	33.36 s.
SA	22.67 s	24.46 s	24.58 s	24.91 s	25.96 s	27.01 s.
FA	22.63 s	24.48 s	25.05 s	25.38 s	26.27 s	27.89 s

Figure 5 is a weak-scaling on MPI ranks. Each time corresponds to the time spent by MPI processes in the factorization, given by the benchmark itself. The efficiency is relative to the mono-rank execution per prioritization policy, this is why times are also given in Table 3. Each MPI process fills a Skylake processor, with 24 OpenMP threads. The scaling starts from a single processor on 1 node, with a matrix factorization of size $n = 41286$, which represents 13% of the node memory capacity. The scaling ends at 16 nodes, with 32 MPI ranks, and a matrix of size $n = 131072$. The exact number of tasks is given in Table 4, where the `compute` category corresponds to `potrf`, `gemm`, `syrk` and `trsm` tasks, and the `communication` category to `send-tasks` and `recv-tasks`.
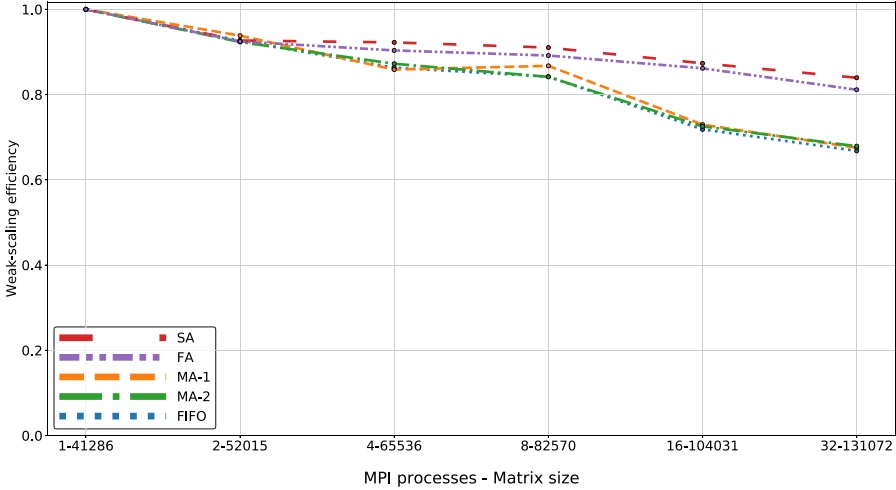
Fig. 5. Weak-scaling on MPI ranks, on the blocked Cholesky factorisation, with blocks of size 512, and MPI processes with 24 OMP threads per rank (Skylake socket)

Table 4. Figure 5 tasking details

Number of ranks	1	2	4	8	16	32
Matrix size (n)	41.286	52.015	65.536	82.570	104.031	131.072
Number of computation tasks	88.560	176.851	357.760	708.561	1.414.910	2.829.056
Number of communication tasks	0	10.100	32.512	102.082	243.616	640.632

In this application, the tasks graph is evenly distributed across MPI ranks. Data-dependencies are whether retrieved from local compute tasks or through `recv-tasks` completion. The weak-scaling result depicted in Fig. 5 scales the number of communications, while keeping constant computation work per MPI-rank. This experiment amplifies the inter-node data exchanges, and thus, the idling phenomenon we have introduced. We see that `MA-1` and `MA-2` prioritization does not improve the performance scaling compared to the reference `FIFO` prioritization. `SA` and `FA` prioritization enables better performance scaling, with up to 19% performance gain in the 32 ranks configuration.

5   Conclusion and Future Work

MPI+OpenMP task programming encounters some interoperability issues that lead to thread idling. Solutions were proposed to address the loss of cores, the load balancing, or the communication collectives, but scheduling issues remain. This paper proposes a task scheduling strategy for OpenMP schedulers to reduce idle periods induced by MPI communications, by favoring `send-tasks`. We propose and evaluate several policies from purely manual approaches which require user cooperation to fully automatic policy.

The best method significantly improves performance and scaling of the Cholesky factorization [19], with up to 19% performance gain in our largest run. Some overhead in the fully automatic strategy has been identified and we are planning to improve graph traversal to reduce the runtime cost.

For future work, we plan to validate our approach on a wider set of applications. Furthermore, in this paper, we only considered explicit data dependencies expressed through the depend clause: we consider adding support for control dependencies. Also, prioritization of `send-tasks` is purely based on local information (the dependency graph between OpenMP tasks) without taking into account task dependencies in other MPI ranks. We are thinking to improve our strategy by taking into account global information.

References

1. Ayguadé, E., et al.: A proposal for task parallelism in OpenMP. In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935, pp. 1–12. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69303-1_1

2. Buettner, D., Acquaviva, J.T., Weidendorfer, J.: Real asynchronous MPI communication in hybrid codes through OpenMP communication tasks, pp. 208–215 (December 2013). https://doi.org/10.1109/ICPADS.2013.39

3. Carribault, P., Pérache, M., Jourdren, H.: Thread-local storage extension to support thread-based MPI/OpenMP applications. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 80–93. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21487-5_7

4. Chatterjee, S., et al.: Integrating asynchronous task parallelism with MPI. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, pp. 712–725 (2013). https://doi.org/10.1109/IPDPS.2013.78

5. Denis, A., Jeannot, E., Swartvagher, P., Thibault, S.: Using dynamic broadcasts to improve task-based runtime performances. In: Euro-Par - 26th International European Conference on Parallel and Distributed Computing. Euro-Par 2020, Rzadca and Malawski, Springer, Warsaw, Poland (August 2020). https://doi.org/10.1007/978-3-030-57675-2_28, https://hal.inria.fr/hal-02872765

6. GNU Project: GOMP - An OpenMP implementation for GCC. https://gcc.gnu.org/projects/gomp/

7. Klinkenberg, J., Samfass, P., Bader, M., Terboven, C., Müller, M.: CHAMELEON: reactive load balancing for hybrid MPI+OpenMP task-parallel applications. J. Parallel Distrib. Comput. **138**, 55–64 (2019). https://doi.org/10.1016/j.jpdc.2019.12.005

8. LLVM Project: OpenMP®: Support for the OpenMP language. https://openmp.llvm.org/

9. Lu, H., Seo, S., Balaji, P.: MPI+ULT: overlapping communication and computation with user-level threads, pp. 444–454 (2015). https://doi.org/10.1109/HPCC-CSS-ICESS.2015.82

10. Maheo, A., Koliaï, S., Carribault, P., Pérache, M., Jalby, W.: Adaptive OpenMP for large NUMA nodes, pp. 254–257 (June 2012). https://doi.org/10.1007/978-3-642-30961-8_720

11. Marjanovic, V., Labarta, J., Ayguadé, E., Valero, M.: Effective communication and computation overlap with hybrid MPI/smpss, vol. 45, pp. 337–338 (2010). https://doi.org/10.1145/1693453.1693502

12. Meadows, L., Ishikawa, K.: OpenMP tasking and MPI in a lattice QCD benchmark. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 77–91. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_6

13. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Spiegel, M., Prins, J.F.: OpenMP task scheduling strategies for multicore NUMA systems. Int. J. High Perform. Comput. Appl. **26**(2), 110–124 (2012). https://doi.org/10.1177/1094342011434065

14. OpenMP Architecture Review Board: OpenMP application program interface version 3.0 (May 2008). http://www.openmp.org/mp-documents/spec30.pdf

15. Pérache, M., Jourdren, H., Namyst, R.: MPC: a unified parallel runtime for clusters of NUMA machines. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 78–88. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85451-7_9

16. Protze, J., Hermanns, M.A., Demiralp, A., Müller, M.S., Kuhlen, T.: MPI detach - asynchronous local completion. In: 27th European MPI Users' Group Meeting, pp. 71–80. EuroMPI/USA 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3416315.3416323

17. Richard, J., Latu, G., Bigot, J., Gautier, T.: Fine-grained MPI+OpenMP plasma simulations: communication overlap with dependent tasks. In: Yahyapour, R. (ed.) Euro-Par 2019. LNCS, vol. 11725, pp. 419–433. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29400-7_30, https://hal-cea.archivesouvertes.fr/cea-02404825

18. Sala, K., Teruel, X., Perez, J.M., Peña, A.J., Beltran, V., Labarta, J.: Integrating blocking and non-blocking MPI primitives with task-based programming models. Parallel Comput. **85**, 153–166 (2019). https://doi.org/10.1016/j.parco.2018.12.008

19. 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.) IWOMP 2018. LNCS, vol. 11128, pp. 3–17. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98521-3_1

20. Seo, S., et al.: Argobots: A lightweight low-level threading and tasking framework. IEEE Trans. Parallel Distrib. Syst. **29**(3), 512–526 (2018). https://doi.org/10.1109/TPDS.2017.2766062

21. Virouleau, P., Broquedis, F., Gautier, T., Rastello, F.: Using data dependencies to improve task-based scheduling strategies on NUMA architectures. In: Euro-Par 2016. Euro-Par 2016, Grenoble, France (August 2016). https://hal.inria.fr/hal-01338761

22. Yang, T., Gerasoulis, A.: DSC: scheduling parallel tasks on an unbounded number of processors. Parallel Distrib. Syst. IEEE Trans. **5**, 951–967 (1994). https://doi.org/10.1109/71.308533