Investigating Dependency Graph Discovery Impact on Task-based MPI+OpenMP Applications Performances

Romain Pereira, Adrien Roussel, Patrick Carribault, Thierry Gautier

Investigating Dependency Graph Discovery Impact on Task-based MPI+OpenMP Applications Performances

Romain PEREIRA
romain.pereira@cea.fr
CEA, DAM, DIF, F-91297
Arpajon, France

Adrien ROUSSEL
adrien.roussel@cea.fr
LIHPC, CEA, DAM, DIF, F-91297
Arpajon, France

Patrick CARRIBAULT
patrick.carribault@cea.fr
LIHPC, CEA, DAM, DIF, F-91297
Arpajon, France

Thierry GAUTIER
thierry.gautier@inrialpes.fr
Avalon, LIP, ENS, Inria
Lyon, France

ABSTRACT

The architecture of supercomputers is evolving to expose massive parallelism. MPI and OpenMP are widely used in application codes on the largest supercomputers in the world. The community primarily focused on composing MPI with OpenMP before its version 3.0 introduced task-based programming. Recent advances in OpenMP task model and its interoperability with MPI enabled fine model composition and seamless support for asynchrony. Yet, OpenMP tasking overheads limit the gain of task-based applications over their historical loop parallelization (parallel for construct).

This paper identifies the OpenMP task dependency graph discovery speed as a limiting factor in the performance of task-based applications. We study its impact on intra and inter-node performances over two benchmarks (Cholesky, HPCG) and a proxy-application (LULESH). We evaluate the performance impacts of several discovery optimizations, and introduce a *persistent* task dependency graph reducing overheads by a factor up to 15 at run-time. We measure 2x speedup over parallel for versions weak scaled to 16K cores, due to improved cache memory use and communication overlap, enabled by task refinement and depth-first scheduling.

CCS CONCEPTS

• **Computing methodologies Distributed programming languages**; • **Computing methodologies Parallel programming languages**;

KEYWORDS

OpenMP, MPI, Task, Dependency, Graph, HPC

ACM Reference Format:
Romain PEREIRA, Adrien ROUSSEL, Patrick CARRIBAULT, and Thierry GAUTIER. 2023. Investigating Dependency Graph Discovery Impact on Task-based MPI+OpenMP Applications Performances. In *52nd International Conference on Parallel Processing (ICPP 2023), August 7–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3605573.3605602

ICPP 2023, August 7–10, 2023, Salt Lake City, UT, USA

1 INTRODUCTION

Supercomputers expose massive parallelism by interconnecting a uniform set of parallel compute nodes. Each node is made of multiple sockets of multi-core processors embedding accelerators. As noted by [1] *"It is therefore critical to find software solutions that can effectively exploit this scale of combined inter-node and intra-node parallelism*". Ten years later, MPI and OpenMP are widely used in application codes on the largest supercomputers in the world.

OpenMP has evolved to version 5.2, and its version 4.0 introduced *dependent tasks* to refine synchronization between tasks accessing the same (declared) memory region. Recent works [2–4] have provided solutions for interoperability issues when nesting MPI communications within OpenMP tasks [5–7], enabling the overlap of communications through OpenMP task scheduling. Yet, few applications have migrated towards such a task-based composition: at least two attempts led to convincing performances with the porting of the Cholesky factorization [6] and the hydrodynamics proxy-app LULESH [8]. Other attempts failed to implement functional applications due to interoperability issues [5], had to tinker application codes [7] to sequentialize communications, or added coarse barriers losing potential communication overlap [9].

A possible explanation may be existing interoperability issues between production runtimes (MPICH/Open MPI/GCC/LLVM) imposing on users to manage an extra non-standard library layer, which could be standardized as proposed in [2, 10]. Moreover as reported in [5], understanding an MPI+OpenMP application behavior is difficult. Task-specific and hybrid visualization tools (Graphs, Gantt charts [11]), debugger and profiler are needed, but none exists supporting the MPI+OpenMP(tasks) composition understudy.

A most possible explanation why such a dependent task-based composition is not widely adopted stems from the performance overheads of handling OpenMP tasks [12]. Consider LULESH [13, 14]: a time step is a sequence of loops which iterate over the mesh data structure. In the OpenMP version provided by LLNL, loops are parallelized using OpenMP `parallel-for` construct. In cases where the workset is large, the multicore execution cannot exploit temporal locality of data. Furthermore, due to `parallel-for` coarse barriers, communication overlap with computation exists but is not expressed in the code, so it cannot be exploited by runtimes.

A contrario, a fine grain dependent task version of LULESH can favor cache reuse with a parallel depth first scheduler such as [15], and exhibits a higher potential of computation concurrent

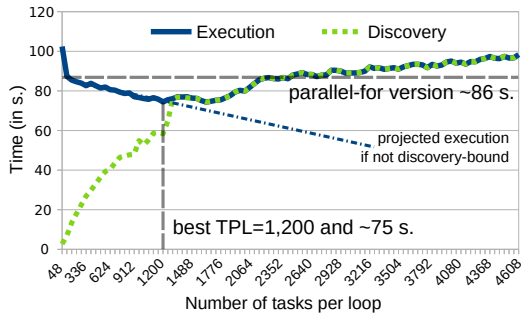

Figure 1: Intra-node LULESH performances with LLVM OpenMP runtime release/16.x on 24 Intel(R) Xeon(R) Platinum 8168 CPU @2.70GHz cores.

to communications. Fig. 1 presents performances using LLVM 16 runtime and filling 78% of the DRAM (`-s 384 -i 16`). Each run is performed on a single process of 24 threads bound 1:1 onto Intel(R) Xeon(R) Platinum 8168 CPU @2.70GHz cores sharing a NUMA domain. The `parallel for` version takes about 86 s. to execute with 98% work time reported by Caliper [16]. From left to right on the task-based version, the number of tasks increases and grains are refined. Fig. 1 reports the time (in seconds) for the task dependency graph *execution* (blue filled curve) and its *discovery* (green dotted curve). The discovery is the time from the first to the last task creation, occuring on a single producer thread concurrently of its execution by any threads (including the producer). The execution corresponds to wall-clock time from the first task schedule to the completion of the last task. It only depends on the scheduler and the architecture: the performance increases while refining tasks, thanks to better data reuse enabled by the depth first scheduler, until the task discovery becomes too slow and bounds the total execution time. Regardless of the number of tasks generated per loop, the performance is at most 6.25% better than OpenMP parallel for loop version, which is not enough relatively to the effort of porting the application to OpenMP dependent task model. On the same experience, GCC runtime did not report any improvements on using dependent tasks.

If we were able to move the cross cutting point of the task graph execution with its discovery along the dashed (blue) curve, the overall performance would be improved. Note that it would not only impact hierarchical memory use, but also in presence of communication, earlier posting and more independent computation discovered for overlapping. This paper presents that accelerating the task dependency graph discovery improves both the work time and communication overlap. The contributions are the following:

(1) a cutting-edge optimized OpenMP tasking runtime,
(2) a new extension to make the task dependency graph persistent against iterations,
(3) an analysis of impact of the task graph discovery on distributed performances over two benchmarks (Cholesky, HPCG) and a proxy-application (LULESH).

It is organized as follows: the next section presents an analysis of intra-node performances of task-based LULESH code. The limiting factor comes from the speed of task dependency graph discovery, which is optimized in section 3. Section 4 studies the impact of these optimizations on three hybrid MPI+OpenMP applications. In section 5, we discuss on the portability of our results across different facets addressed by HPC developers using MPI + OpenMP. We then review related works before providing conclusion and perspectives.

2 MOTIVATION

Let us introduce the motivation of our work through the LULESH case study. The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) is a proxy application representing the computation kernel of hydro-simulations, originally provided by LLNL for co-designing towards exascale. As almost every CORAL benchmark, LULESH adopts the MPI+X programming paradigm: data are distributed among nodes of the supercomputer with MPI, and shared-memory parallelism is adressed with OpenMP `parallel for` loops. The application simulates on an unstructured mesh with $O(s^3)$ nodes over i time-step iterations. Parameters s and i can be passed through command line arguments. The simulation is a sequence of mesh-wide computational loops, MPI point-to-point communications with neighbors mesh connected through nodes, edges or faces, and an MPI collective communication reducing the dynamic time step at the end of each iterations. The proxy-application comes with two reports [13, 14] that present how the source code should be used to remain representative of HPC scientific simulation needs. For instance, constraints include the mesh data structure representation, loop structures, and extra computation. Every version presented in this paper respects these constraints. This section is devoted to preliminary analysis on the impact of the *Task Dependency Graph* (TDG) discovery on the performance with a focus on multi-core architecture.

2.1 Reference Parallel-for version

The original LULESH code from LLNL uses a *fork-join* programming model. The computational loop sequences are parallelized with OpenMP `parallel for` construct and data are distributed with MPI communications outside of OpenMP constructs. The original code performances are rather poor as studied in [17]. Therefore, we backported the *global allocation of temporary work arrays* optimization, which increases performances by 24% on our configuration and is compatible with reports constraints.

Such a programming approach limits parallelism due to coarse synchronizations. *Load balancing* is limited per-loop, while iterations of separate loops could run in parallel independently. Additionally, the code expresses a rigid sequence of parallel computations due to barriers induced by OpenMP `parallel-for` loops semantic. Temporal data reuse between loops cannot be achieved with the limited size of hardware caches. Moreover, potential *communication overlap with computation* exists, but it is not expressed in the code due to `parallel-for` coarse barriers, and therefore cannot be exploited by the runtime. For instance, inter-process communications could start as soon as nodes connecting mesh frontiers have been computed instead of waiting for the entire local mesh computation.

2.2 Task-based version

To overcome the OpenMP `parallel-for` drawbacks, one might consider using dependent task-based programming models which can express a higher potential of asynchronous computation and communications, following the application data flow.

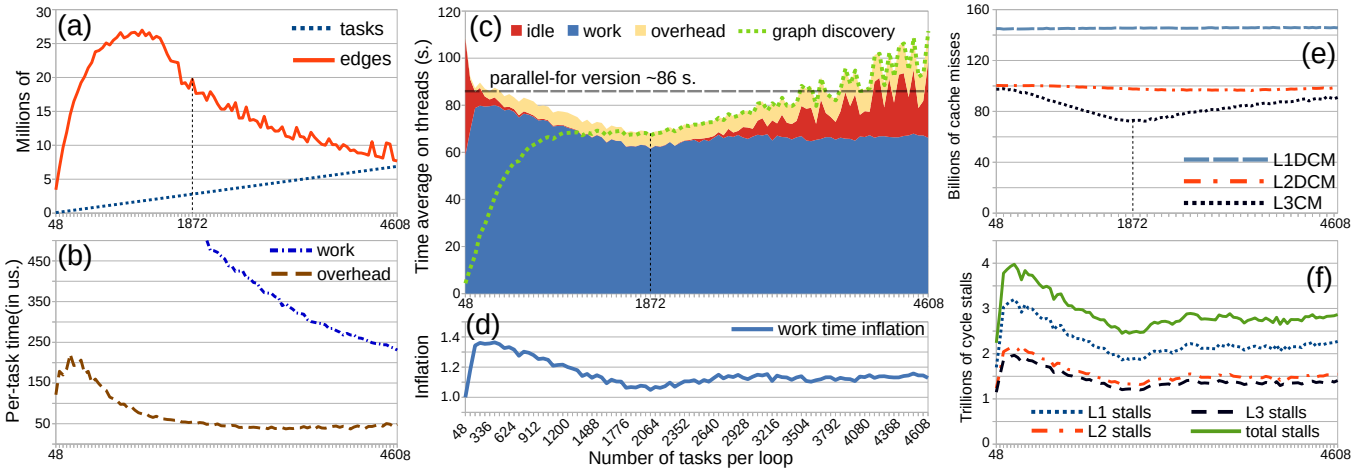

Figure 2: LULESH on a 24-cores Skylake node using MPC-OMP with -s 384 -i 16 -tel tasks-per-loop

Thanks to the authors, we retrieved a task-based LULESH implementation developed by Ferat et al. [8] [1]. To fully respect the mentioned constraints, it relies on using the OpenMP depend clause on the `taskloop` construct [18] which is not standard yet. In addition, the code uses the `inoutset` dependence-type recently introduced in version 5.1 of the OpenMP specification, which not yet available in production OpenMP compiler/runtime releases such as GCC, LLVM or ICC. Therefore, the code is relevant to very up-to-date OpenMP task based programming.

Its structure is the following and shown on Listing 1. The entire application is wrapped into an OpenMP `single` region and replaces all `parallel-for` constructs with dependent tasks generated by taskloop. The correct order of execution is ensured through dependences built upon read and write accesses on the mesh nodes and elements. The computational part of the code comes from the original version of the benchmark. A *Tasks Per Loop* (TPL) parameter defines the number of tasks on mesh-wide loops: tasks grain and dependences are automatically infered from it. This parameter modulates the number of tasks to find a compromise between the parallelism expression and induced overheads.

MPI communications were placed into OpenMP tasks and are inserted into the TDG as any other tasks. It enables communication overlap with independant tasks, but also earlier request posting as communications tasks only depends on on frontier nodes computation, as opposed to the `parallel-for` version where the entire domain must be computed before initiating communications.

```
1  double dt;
2  for (int iter = 0 ; iter < max_iter ; ++iter)
3  {
4      # pragma omp task depend(out: dt)
5      {
6          double local_dt = compute_local_timestep();
7          MPI_Allreduce(&dt, &local_dt, sizeof(double, MPI_MIN,
               MPI_COMM_WORLD));
8      }
9
10     # pragma omp taskloop depend(in: dt) depend(out: mesh.nodes[i])
           firstprivate(iter) num_tasks(t)
11         for (int i = 0 ; i < mesh.nodes.size() ; ++i)
12             work_on_nodes(mesh.nodes[i], dt);
13
14     # pragma omp taskloop depend(in: dt, mesh.nodes[...]) depend(out:
           mesh.elements[i]) firstprivate(iter) num_tasks(t)
15         for (int i = 0 ; i < mesh.elements.size() ; ++i)
16             work_on_elements(mesh.elements[i], dt);
17
18     // more domain-wide loops
19
20     # pragma omp task depend(out: rbuffer) detach(event)
21         MPI_Irecv(..., rbuffer, ...);
22
23     # pragma omp task depend(in: mesh.nodes[...]) depend(out: sbuffer)
24         Pack(sbuffer, mesh);
25
26     # pragma omp task depend(in: sbuffer) detach(event)
27         MPI_Isend(..., sbuffer, ...);
28
29     # pragma omp task depend(in: rbuffer) depend(out: mesh.nodes[...])
30         Unpack(rbuffer, mesh);
31
32     // more domain-wide loops
33 }
```

Listing 1: Task-based LULESH code structure

2.3 Performances Analysis

We reproduced the experiment presented Fig. 1 with LLVM on Fig. 2 using the MPC OpenMP runtime: MPC-OMP. It is a standard-compliant runtime publicly available [2]. It features capability to mix with any MPI implementations, polling MPI requests on OpenMP scheduling points. It uses a LIFO and depth-first scheduling heuristic, and showed similar performance if not slightly better than the LLVM runtime. Optimizations presented in Section 3 were implemented in the MPC-OMP runtime.

2.3.1 Methodology. We built a profiler within MPC-OMP for reporting metrics of task-based executions. The profiler traces tasks

[1] https://github.com/rpereira-dev/LULESH

[2] https://mpc.hpcframework.com/download/

schedule, creation, and dependencies. Each event record are written to preallocated memory region in DRAM during the execution, and flushed to disk on termination. They are timed using `omp_get_wtime` routine under a microsecond precision. Configurable hardware counters can also be attached to records with PAPI [19], allowing fine performance characterization at the task-level, such as work time inflation due to non-uniform memory accesses described in [20]. Such profiling approach can impact the performances and is limited by the DRAM capacity. In the profiled results of this paper, we ensured that the trace fits into the DRAM and observed from 0% to 5% performance degradation over non-profiled execution depending on the number of tasks.

We also implemented post-mortem analysis to compute performance metrics. We adapted the parallel time breakdown proposed in [21] to dependent tasking model: the *work* is the time spent within a task body, the *overhead* is the time spent outside of a task body while there is tasks ready, and the *idleness* is the time spent ouside of a task body while there is no tasks ready. These times are cumulated and averaged on cores.

2.3.2 Overview of the results. Fig. 2 depicts results from a profiled execution of the experiment Fig. 1 with MPC-OMP. Fig. 2 (a) represents the overvall number of tasks and edges discovered for the given TPL, and sugfigures (b) depicts the average overheads and work time per task. As shown on Fig. 2 (a,b), the right-most point reaches a total number of 7,5M OpenMP tasks with an average grain of 250μs. Fig. 2 (c) depicts the time breakdown averaged on threads. The bottom blue stack is the work time, the middle red stack the idle time and the top yellow stack the overhead. The green dotted line represents the TDG discovery time on the single producer thread. Fig. 2 (d) shows the work time inflation of each TPL instances using the less inflated TPL instance as a reference. Fig. 2 (e,f) respectively depicts the number of misses and stalled cycles per cache-level, occuring when cores execute task work.

2.3.3 TPL analysis.

Coarse grain. The left-most point is the less inflated because LULESH is memory-bound by the DRAM bandwidth and having only 48 tasks per loop reduces the amount of parallel work and thus, memory contention. This can be seen on Fig. 2 (c) where cores spend a lot of time idling due to low parallelism. In average, it leads to reducing DRAM contention and accelerating the work. Note that the overall execution time still ends up slower due to idleness.

Middle grain. From 192 to 1,872 TPL, on Fig. 2 (d) the work time deflates significantely from 40% inflation to 10%. Fig. 2 (e,f) provide explanation: on this TPL range, we observe a reduction of L3 cache misses (L3CM) and stalled cycles due to cache-miss. Whenever a data access causes a L1DCM or a L2DCM, the memory controller will more likely find the data in the L3 memory without having to retrieve it from the DRAM. It accelerates memory access with less process stalls, hence the observed work time deflation. This better use of the memory hierarchy is the results of two tasking mechanisms: (1) tasks refinement reduces average data size accessed per task and (2) the depth-first scheduling heuristic favors the execution of tasks' successors improving cached-data reusability.

Fine grain. After 1,872 TPL, the application execution time is bound by the TDG discovery time. Many edges are pruned because tasks are getting consumed as soon as they are produced and no longer exists on their successors' discovery, as it can be seen on the subfigure (a). This is also reflected by an increase in the idle time on the subfigure (c) where threads ends up idling waiting for tasks to spawn. Being *discovery-bound* limits the vision of the TDG to the OpenMP scheduler: the depth-first scheduling heuristic cannot be effective because successors are not known by the scheduler on predecessors completion. Threads end up scheduling whatever is ready (in a *breadth first* manner) resulting in poor cache reusability.

Instance	Idle (s.)	Work (s.)	L2DCM	L3CM
1,872 TPL Normal	3.5	1,477	98B	73B
4,608 TPL Normal	766.6	1,589	98B	91B
4,608 TPL Non overlapped	1.2	1,077	84B	53B

Table 1: Impact of the task graph discovery on the work time

2.3.4 Impact of the TDG discovery on the work time. Overlapping TDG discovery with its execution has an impact on task scheduler that promotes data reuse: if the TDG discovery is too slow, the data-producing task will never activate its successor as it has not been discovered yet. To measure the impact of the TDG discovery on its execution time, we run a complementary experiment blocking execution by threads until the TDG has been fully discovered. The MPC-OMP scheduler will have access to the description of all the dependencies before taking decisions.

Table 1 shows the cumulated work and idle time on threads, the number of L2 data cache misses and L3 cache misses on different configurations. The instance with tag *Normal* corresponds to execution with TDG execution and discovery overlapping. Instance with tag *Non overlapped* means the graph is first fully discovered before starting its parallel execution.

The two first rows present results from the best (1,872 TPL) and finest (4,608 TPL) grain execution shown on Fig. 2, under the Normal configuration. The third row presents results with 4,608 TPL with non-overlapped configuration. Comparing the two configuration with 4,608 TPL, in-depth knowledge of the TDG leads to significantly reducing the cache misses in L2 (−15%) and L3 (−42%) for a 32% work time reduction. We also observe almost no idleness as threads do not have to wait for tasks to spawn. Work time and idleness gain leads to a 40% parallel execution time reduction. Though, the total execution time is still much slower in the non-overlapped configuration because the entire graph must be unrolled sequentially first: 357s with the non overlapped experiment and 112s for the normal execution.

2.4 Summary

In this section, we presented an in-depth analysis of the gains of a task based LULESH against the `parallel-for` reference version. Data reuses increase on high TPL value. We measure a minimal time and work time at TPL=1,872 which is much bigger than the number of cores (24). The task-based version executes in 69s with MPC-OMP against 86s on the reference version using LLVM 16. Moreover, we also report that accelerating the TDG discovery on the task-based version would allow reaching finer task grain so accessed data fitting into the L2 cache level. It would also enable

effective depth-first over breadth-first scheduling, and less idleness for a 40% gain on the parallel time.

3 ACCELERATING TDG DISCOVERY

TDG discovery speed is a limiting factor to reach high performances. The purpose of this section is to present optimizations on OpenMP TDG discovery related to runtime and user codes. Some of them are standard, but we also propose an extension caching internal tasks data structure with very light intrusive cost in term of code modification. All together, optimizations accelerate the TDG discovery and we reports their impact on performances in section 3.3.

3.1 Reducing the number of edges

An edge corresponds to a precedence constraint between two tasks. Because of the sequential task submission and the very *local* view (per-procedure) of tasks, some unnecessary edges may appear between tasks which order of execution is already guaranteed from other edges. To accelerate the TDG discovery, we present three optimizations to remove unnecessary edges.

The optimization (a) is depicted in Fig. 3. It consists in minimizing the number of dependences expressed from the *user code* preserving both the parallelism and the correct order of execution. In this example, the task line 1 writes (x, y) which are only read by the task line 8. This provides two data addresses to be processed by the runtime while only one is really needed. Such dependency pattern appeared within Ferat et al. LULESH and were optimized.

```
1 # pragma omp task depend(out: x, y)
2 {
3     x += dx * dt;
4     y += dy * dt;
5 }
6
7 # pragma omp task depend(in:  x, y)
8     printf("%f %f\n", x, y);
```

Figure 3: Optimization (a) - multiple edges

The optimization (b) consists in automatically removing multiple edges by the runtime. On edge creation, the runtime can detect multiple edges with O(1) time-complexity thanks to the sequential submission of dependent tasks. Note that optimization (a) differs from (b) in the sense that it not only deletes multiple edges but also cost of detecting them in (b). The optimization (b) is implemented into GCC but not into LLVM. We implemented it in MPC-OMP.

The optimization (c) is related to the `inoutset` dependency type. Tasks t_i having a depend clause of type `inoutset` on the same data can run concurrently but successor tasks with any other dependency type on the same data will depend on every t_i task. This dependency type is also known as *concurrent write* in Athapascan [22] or OmpSs [23] and is to be used when multiple tasks can write concurrently in a memory block, giving this knowledge to the runtime for optimizations. Fig. 4 depicts a minimal example of an `inoutset` dependency scheme where m tasks $(X_i)_{i\in[1,m]}$ concurrently write onto the vector x which is read by the n tasks $(Y_j)_{j\in[1,n]}$. The optimization (c) consists in inserting an extra empty node R by the runtime within the TDG to reduce the number of edges from $m.n$

```
1 for (int i = 1 ; i <= m ; ++i)
2 {
3     # pragma omp task depend(inoutset: x)
4         x[i] = i;
5 }
6
7 for (int j = 1 ; j <= n ; ++j)
8 {
9     # pragma omp task depend(in: x)
10        work(x, y[j]);
11 }
```

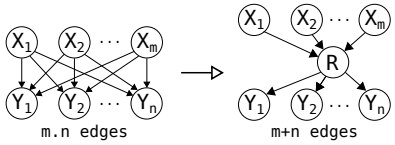

Figure 4: Optimization (c) - inoutset edges reduction

to $m + n$. This optimization is implemented in LLVM [3] but not into GCC. We implemented it in MPC-OMP.

3.2 Persistent Task Sub Graph (PTSG)

LULESH is an iterative application onto an unstructured mesh. On each iteration, the same task dependences are rediscovered built upon mesh' nodes and elements. To accelerate the discovery

```
+1 # pragma omp ptsg
 2 for (int i = 0 ; i < iterations ; ++i)
 3 {
 4     # pragma omp task depend(in: x) firstprivate(i)
 5     { }    /* some work */
 6
 7     [...]
 8 }
```

Figure 5: Optimization (p) - persistent graph annotation

furthermore, the optimization (p) reduces overheads by adding persistence to task dependences. It is illustrated in the Fig. 5. From users point of view, it only consists in annotating a loop generating dependant tasks in the same order and with the same dependency scheme on each iteration.

Implementation. On the first iteration, the runtime discovers the TDG just like before but internally marking tasks as *persistent*, so they are not destroyed on completion. It also creates every edge, as opposed to non-persistent mode which prunes edges to tasks already consumed, making the first iteration of persistent task graph discovery more costly. Creating every edge is necessary since no edges are recreated on future iterations, to ensure the correct order of execution with no race conditions. Then on future iterations, the producer thread only copies tasks `firstprivate` data such as the loop iterator variable. It relieves the runtime from other tasking overheads sources such as the internal task descriptor allocation, dependences processing (depend clause) or Internal Control Variable (ICV) management. Therefore, task initialization cost is reduced to a single `memcpy` on `firstprivate` data, representing 8 to 100 bytes in LULESH tasks. An implicit barrier at the end of each iteration ensures that every task is completed before re-instancing them.

[3] https://reviews.llvm.org/D97085

optimizations	n° of tasks	n° of edges	discovery (s.)	execution (s.)	work (s.)	idle (s.)	overhead (s.)
none	2,802,400	56,320,296	76.67	77.48	1,612	3.49	229.50
(a)	2,802,400	58,168,382	71.79	73.40	1,533	3.00	211.40
(b)	2,802,400	21,926,084	67.53	71.83	1,546	2.85	131.70
(c)	4,853,811	30,246,767	71.91	73.19	1,568	3.07	167.44
(a)+(b)	2,802,400	31,144,779	66.60	69.39	1,503	3.23	144.26
(a)+(c)	3,614,124	36,239,253	56.38	69.43	1,512	3.31	128.51
(b)+(c)	4,858,307	24,837,165	71.73	72.95	1,568	2.85	163.29
(a)+(b)+(c)	3,613,737	27,633,254	55.56	69.17	1,517	2.64	116.77
(a)+(b)+(c)+(p)	3,557,840	19,177,294	5.97	70.80	1,655	12.22	27.30

Table 2: Graph optimizations crossing

Impact on codes, runtimes, and standard. The impact of persistent TDG on user code is lite: from given task-based applications, a single line of code (LoC) annotation on a loop enabled persistence. Our implementation had a moderate impact on OpenMP runtime, adding only 175 LoC but changing critical and highly concurrent runtime code related to tasks' lifecycle. The standardisation of *persistent* task dependency graphs raises multiple issues. A `taskgraph` [24] proposal was recently made. However, it does not provide fine control on task attributes: everything is assumed persistent. They also only provided specifications but no runtime implementations. In our implementation, we allowed the update of `firstprivate` data dynamically without invalidating the entire graph: Shared data, dependences or ICVs could also be updated for instance, and standardization is kept as future work.

Applicability. The main performance gain we obtained from persistent graph is from dependency management. This is possible because our approach assumes dependences between tasks to remain constant over iterations. However, on simulation over unstructured meshes like LULESH, Adaptive Mesh Refinement (AMR) may occur during the simulation, slightly changing the mesh and so, the TDG. Nevertheless, because of the inherent costs of mesh adaptation, applications try to amortize it over a few iterations. In that way, our proposal would benefit from this already existing strategy.

3.3 Evaluation

Table 2 depicts the number of edges, the TDG discovery, the total execution time, and the work/idle/overhead time breakdown, crossing each optimization on the same problem as of Fig. 2 with $TPL = 1,872$; for about 2.9 M. tasks of $365\mu s$ grain in average.

Comparing (b) with (a)+(b), more edges are generated in the second case even though TDGs discovery are about the same. This phenomenon occurs because when enabling optimizations, the TDG discovery is faster, leading to more predecessors existing on successors' creation; therefore, less automatic pruning occurs. Though, enabling all optimizations (a)+(b)+(c) led to 2.04 fewer edges and 1.38 speedup on the discovery over the non-optimized version.

As shown on (a)+(b)+(c) and (a)+(b)+(c)+(p), enabling tasks' persistence optimization (p) drastically reduces the discovery time by an order of magnitude. Among the $5.97s$ of discovery, $4.11s$ correspond to the first iteration discovery, and the remaining corresponds to the 15 other iterations taking $0.12s$ in average: the first iteration is about 34 times slower than the others, as it is the one responsible for building the dependency graph while the others simply update tasks private data. The first persistent iteration ($4.11s$) is more costly than non-persistent iterations ($3.47s$ in average), and even though every edges are created (edges to tasks already consumed are not pruned with persistent tasking), we observe less edges enabling (p). This is a side effect of our runtime implementation of persistence, which has an implicit barrier on each iteration, removing inter-iteration edges.

Moreover, this barrier leads optimization (p) to increase the total execution time ($69.17s$ to $70.80s$): the work and idle times increase because the barrier prohibits successor tasks of iterations $n+1$ from starting until every task of iteration n is completed. This is shown on the Gantt chart Fig.8 where one color represents one iteration. Threads spend more time idling in this synchronization barrier, and the work time inflates as depth-first scheduling is constrained per iteration. However, as (p) accelerates the graph discovery, overheads are reduced, enabling effective depth-first scheduling at a much finer grain.

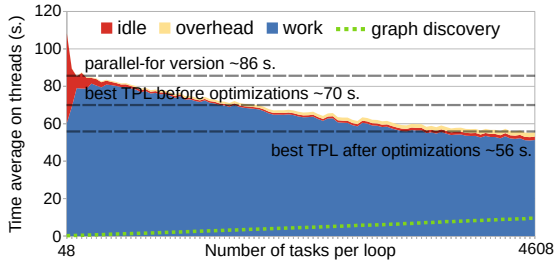

Figure 6: LULESH on Intel Skylake node with optimizations

Impact on Shared-memory Execution. Fig. 6 depicts the same time breakdown as in Fig 2 but enabling every optimization. The TDG execution is no longer bound by its discovery and enable effective depth-first scheduling. It leads to 1.56x speedup over the parallel-for version and 1.27x speedup over the non-optimized task-based version. The 4,608 TPL configuration reaches a $1,230s$ work time for $82B$ L2DCM and $54B$ L3CM.

METG report. The Minimum Effective Task Granularity metric was proposed [12] as a way to assess on the overheads of tasking runtimes. For a given application and runtime system, $METG(X\%)$ gives the minimum task grains for which an instance of execution reaches $X\%$ of the best performances measured on any runtime system. Their results on OpenMP shows a $METG(95\%) = 1ms$ for several applications. Running LULESH with GCC, LLVM and MPC-OMP tasking runtime, we measured an $METG(95\%)$ of $65\mu s$ with

9,216 TPL using MPC-OMP, which is 1.5 order of magnitude less than the best OpenMP METG reported in [12] for such efficiency.

4 IMPACT ON DISTRIBUTED EXECUTION

The TDG discovery speed also impacts distributed execution for application with MPI and OpenMP dependent tasks. We assemble applications (LULESH, HPCG, Cholesky) with different parallel characteristics (computation, communication) that will lead to three conclusions. On considered applications, MPI communications are nested into OpenMP task region as permitted by recent published results on MPI interoperability [2, 4, 10, 25, 26]. We performed distributed performance evaluations on AMD EPYC 7763 64-core CPU, interconnected with Atos BXI V2 running Open MPI 4.1.4. In our experiment, we place MPI processes per NUMA domain with 16 OpenMP threads compactly bound 1:1 to cores. LULESH was scaled to fill 72% of each NUMA domain on this new architecture (`-s 256`), running on 16 nodes with 125 MPI processes.

4.1 Distributed LULESH Performances

LULESH comes with 3 types of point-to-point (P2P) requests communicating mesh frontiers with neighbors: a single node, an edge, or a face, respectively with O(1), O(s) and O(s^2) bytes transfers. For our given problem size and our Open MPI configuration, O(1) and O(s) bytes MPI requests are performed with a eager protocol but O(s^2) requests are performed using rendezvous.

As opposed to the `parallel-for` version, which implies entire mesh-wide O(n^3) computation to complete before starting communications, the task-based version allows posting of MPI requests as soon as predecessor tasks working on frontier nodes have completed. Hence, a depth-first scheduling strategy can lead to earlier communication posting, and preserve independent work for overlapping communication, which is the subject of this study. Fig. 7 presents performances on an MPI process connected whose mesh is connected to 26 other MPI process meshes. It shows the time breakdown and the communication time for the `parallel for` version (LLVM 16), the non-optimized task-based version (MPC-OMP), and the optimized task-based version (MPC-OMP).

Computational Performances. Work/idle/overhead times were retrieved using the methodology Section 2.3.1 for MPC-OMP and Caliper for LLVM (which only provides work/non-work times). We observe the same performance gain as on the previous architecture. The optimized task-based is 2.0x and 1.2x more performant than the `parallel-for` and the non-optimized task-based version, for the exact same reason of hierarchical memory accesses. Note that we observe work time deflation on the non-optimized task-based version after 2,176. Above this threshold, the TDG discovery becomes too slow to feed every core (as shown by the important idleness), reducing the DRAM contention, which in turn, accelerates memory accesses of the few threads working in parallel.

Methodology on Communications Profiling. We extended the profiler presented in Section 2.3.1 with PMPI to support MPI communication profiling and analysis. Our analysis only consider *send* and *collective* requests and comes with three metrics: the communication time, the overlapped work, and the overlap ratio. Given an MPI request r, the communication time $c(r)$ is the duration from r posting (`MPI_Start`, `MPI_Isend`, `MPI_Iallreduce`) to r completion (success on `MPI_Wait` and `MPI_Test`). The overlaped work $ov(r)$ is the sum of work occuring in parallel of the communication time on any local cores (16 in our evaluations). The communication and overlapped work are then obtained by reducing on every MPI request, as $C = \sum c(r)$ and $W = \sum ov(r)$. Finally, we define the overlap ratio as $r_{overlap} = \frac{W}{\text{n-threads} \times C}$. This definition extends the usual single-thread overlap measurements to multi-threading: n-threads $\times C$ corresponds to an ideal *overlapable* work time on the multi-threaded MPI process during the communication progression.

Communication Overlap with Computation. The `parallel-for` version exhibits no overlap potential on P2P send communications. All the requests are posted in non-blocking mode, and the execution flow waits for every completion before pursuing any computation. The collective is in blocking mode, posted at the beginning of a new iteration. The collective of iteration $n+1$ depends on every task of iteration n. However, the Gantt chart of the task-based version on Fig. 8 shows that there is some independent computational tasks of iteration $n+1$ such as `CalcFBHourglassForceForElems` ready for overlap. On the `parallel-for` version, all this work would have been processed synchronously after the collective operation following the sequential instruction flow.

On the task-based versions, the overlap ratio is improved when enabling optimizations, and remains above 80% on any TPL against 50% in average without optimizations as shown Fig. 7: the application discover tasking parallelism faster and more work is ready for overlap masking communication costs.

To estimate the gain from integrating MPI communications into OpenMP TDG following the data flow, we added explicit `# pragma omp taskwait` before and after communication sequences. With the tracing disabled, and on the most performant configuration (TPL=4,608), we measured 131.0s with taskwait against 121.9s with no taskwait: fine integration of MPI communications into OpenMP TDG reduced by 7% the total execution time.

Communication Time. We made a breakdown of communication time of each TPL, and in average, 94% of the communication time corresponds to the MPI Iallreduce collective, and the remaining 6% are the 26 P2P send communications. The variation observed on the communication is therefore mostly related to this single collective.

Gantt charts Fig. 8 shows the execution of tasks from iterations 11 to 15 on the two task-based version of $TPL = 1,152$ of Fig. 7. Boxes represent task schedules, and each color identifies a different iteration. Time origins had been offset on each chart along the x-axis, to align to the first task of the iteration 12 (green). Because of the implicit task barrier on each iteration induced by our persistent TDG implementation, no tasks from the iteration $n+1$ can start until every task from iteration n completed (bottom chart). Looking at the two charts, this behavior seems to inflate collective synchronization time globally. This phenomena likely explain why the the (collective) communication time is faster on the non-optimized version on $128 < TPL < 1,280$.

Refining from 128 to 1,280, we also observe a reduction on communication time. We believe it comes from faster local execution (as shown on the time breakdown), allowing in average earlier collective communication matching on every MPI process.

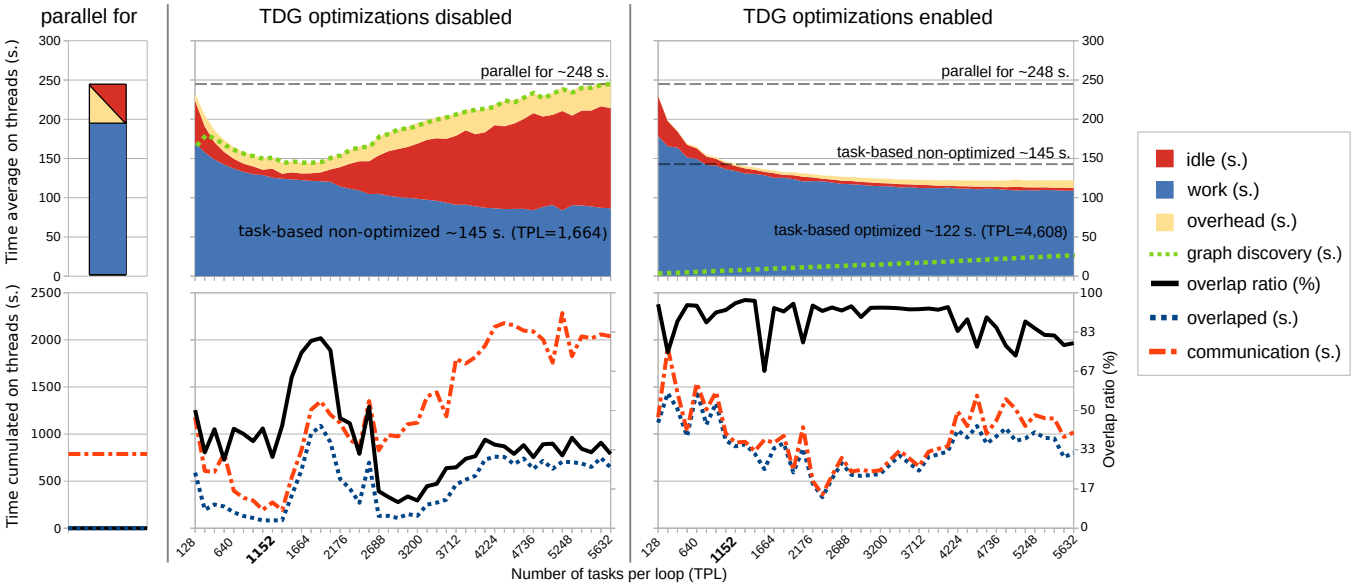

Figure 7: LULESH -s 256 -i 64 on 125 MPI processes, 54 AMD EPYC 7763. Time breakdown (top) and communication performances (bottom) on a profiled MPI process (rank 82)

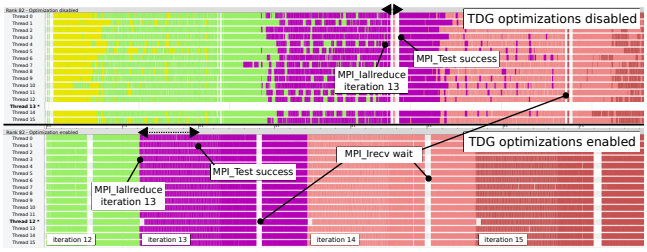

Figure 8: Gantt chart of task-based execution for TPL=1,152

However, refining furthermore on the non-optimized version leads to important idleness, as the TDG discovery becomes too slow to feed every core with work. This may explain the communication time slowdown: every MPI process must wait for the slowest local OpenMP TDG discovery. Results show that accelerating the TDG discovery lead to a more stable communication time at fine grain.

4.2 Scaling to 65k cores

We performed a strong and a weak scaling on LULESH for both the `parallel-for` (GCC 11.2.0) and our optimized task-based version (MPC-OMP). We increased the number of simulation iteration from 64 to 1,024. Table 3 presents wall clock time results scaling from 8 MPI processes (= 1 node) to 4,096 processes on a single run. No performances could be recorded on the weak-scaling above 1,331 processes because all versions abort on a numerical error.

From 8 to 1,000 MPI processes with 2,048 TPL, task-based executions lasted about 2,000 s. with more than 95% weak-scaling efficiency, and a 2.0x speedup compared to the `parallel-for` version. The strong-scaling from 8 to 4,096 MPI processes use a dynamic TPL to balance parallelism and workload per tasks, ensuring at least 16 tasks per loops and at most 8,192 mesh nodes per tasks, as shown in the last row. Performance improves over the `parallel-for` version until 128 MPI processes for about 5% DRAM use; then, fine grain execution provide no gain.

4.3 HPCG

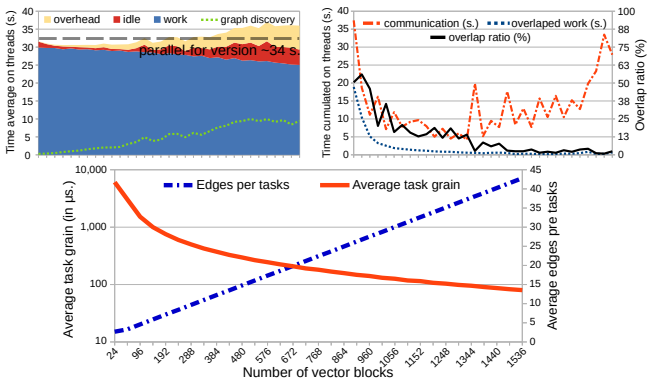

Figure 9: HPCG performances on 32 MPI processes on Intel processors for a matrix with n=41,943,040 on i=128 iterations

The High Performance Conjugate Gradient (HPCG) is a benchmark used to rank supercomputers as a complement to the LINPACK (HPL) benchmarks [27]. The baseline implementation is parallelized using `parallel-for` construct and barriers before communicating with MPI. We ported HPCG to using OpenMP dependent task model with two grains parameters defining the number of block for vector-wise operations (TPL) and the number of sub-blocks for SpMV operations, that we fix to 32 in future experiments. Fig. 9 present our results on 32 MPI processes of 24 threads varying the TPL parameters, on the Intel Skylake processors of Section 2.

On Communications. Depending on the TPL, the communication time varies from 5 to 37 s. for a cumulated work time around 700s.

MPI processes	8	27	64	128	216	343	512	729	1,000	1,331	1,728	2,197	2,744	3,375	4,096
weak - for (s.)	3,926	3,935	3,953	3,954	3,979	4,006	4,012	4,141	4,181	N/A	N/A	N/A	N/A	N/A	N/A
weak - task (s.)	2,065	2,136	2,074	2,088	2,153	2,077	2,093	2,185	2,089	N/A	N/A	N/A	N/A	N/A	N/A
strong - for (s.)	3,926	895	305	150	85	61	44	30	19	15	11	10	8	10	11
strong - task (s.)	2,065	627	267	148	88	69	49	43	29	23	16	16	11	13	9
strong - TPL	2,048	599	256	129	74	47	32	21	16	16	16	16	16	16	16

Table 3: LULESH -s 256 -i 1024 - Weak and Strong scaling from 8 to 4,096 MPI processes

It means with perfect overlap, at most 5% of the work time would be overlaping communication. Moreover, even though TDG discovery is fast-enough, the overlap ratio remains low ($\leq$ 23%) meaning little work is available in parallel of communications. Therefore, there is little no to gain to expect from overlapping communication with computational tasks on HPCG.

Time Breakdown. Regarding work time, the best performances are reached for an average tasks grain of 80μs using the right-most 1,536 TPL. It allows 20% work time reduction compared to the baseline parallel-for version. Reasons are the same as LULESH: memory accesses are faster due to better cache reuse. Even though TDG discovery optimizations were enabled, fine grain tasks management overheads deteriorate total execution time more than the work time gain. Though, the minimal total time (30.6s) is obtained with TPL=144 (1ms/tasks) for 10% performances gain over the `parallel for` version with LLVM 16 (34.1s. with 95% work time). Above this grain, overheads and idleness deteriorates more performances than the work time improvement. We explain this by the *tasking runtime contention*: as shown on Fig. 9, the number of edges per tasks grows linearly from 24 to 1,536 TPL while the workload per task decreases. Runtime-side, this is reflected by more threads accessing more often shared data structure, such as the task dependency graph.

4.4 Tile-based Cholesky Factorization

Tile-based Cholesky dense matrix factorisation is a widely studied application of dependant task-based programming. We retrieved the version of [6] with dependent task and MPI communications performed by tasks. Optimizations (a), (b) and (c) does not provide any performance improvement/degradation as they are not reflected in the application: its dense dependency scheme is simpler than sparse and indirection-based data structures found in HPCG and LULESH. The optimization (p) provides performance gain on TDG when iteratively decomposing matrices of same dimensions and tile size. We evaluated the TDG discovery speed gain for a matrix of size $n = 65,536$ with block sizes $b = 512$, distributed on 32 MPI processes of 24 cores over 16 Intel nodes, same one used in Section 2. Our results showed 5x asymptotical speedup on the discovery when increasing the number of iterations. It showed no significant impact on performances as the TDG discovery is already fast due to coarse tasks and regular dependencies (<2% of total time). On 16 Skylake nodes (768 cores), we measured 269s (with) and 274s (without) optimizations for matrix n=65,536 with block-size m=512.

4.5 Summary

In this section, we analyzed performance gain induced by accelerating the TDG discovery on three applications. Our best result shows a 2.0x speedup on LULESH using OpenMP tasking over `parallel-for` weak-scaled to 16,000 cores, thanks to task grain refinement and depth-first scheduling. Finely composing MPI and OpenMP by integrating communications into dependent tasks lead to 7% execution time reduction. HPCG evaluations reports moderate performances with a 1.1x speedup, due to difficulties on balancing work time gains with runtime contention. Cholesky results do not provide neither performance improvements nor degradations.

5 DISCUSSIONS

Our results show a 2.0x speedup on the task-based version of LULESH over the parallel-for/BSP one. These performances were reached after accelerating the TDG discovery, enabling fine task grain depth-first scheduling. It led to better use of the memory hierarchy and overlapping of communication with computation. Nevertheless, the level of performances of distributed task based execution is only possible if the OpenMP runtime does not restrict the graph discovery and interoperate with the MPI layer.

Task Throttling. Task Throttling is a runtime mechanism to reduce tasking operational and memory overheads [28]. Once a threshold is reached, producer threads stop producing and start consuming tasks instead. This mechanism limits the vision by the runtime scheduler of the future of the execution [7].

Both GCC and LLVM runtimes implements a threshold bounding the number of *ready-tasks* that can co-exist. In the context of OpenMP, task throttling was developed for independent task model (OpenMP-3.0) as an efficient solution to bound the memory consumption [28]. Note that in case of dependent tasks, ready-tasks throttling threshold is not sufficient: as many successors tasks may be created but not marked as ready. Therefore, the MPC-OMP runtime implements in addition a *total tasks* threshold bounding the total number of co-existing tasks (ready or not), with a parametrable value[4]. Our evaluations showed best performances on LULESH with $3,072$ tasks per loop on 33 loops, which represents around 100,000 tasks per simulation iterations. Even with faster TDG discovery, GCC/LLVM runtimes would not benefit from finer tasks and depth-first scheduling as their task throttling implementation would not allow in-depth vision of the TDG. Though, throttling can be fully disabled in LLVM [5] making memory usage bounds a user responsibility. Finer control should be added to production OpenMP runtimes to ensure bounds on both parallelism and memory usage for dependent tasking.

Dependency Processing. The optimizations studied Section 3 illustrates that TDG discovery is a shared responsibility between user codes and runtimes, which the standard interface can orchestrate. The recent proposal on adding a `depend` clause on the `taskloop` construct [18] is necessary to preserve existing code loop structure, and may open to new compiler/runtime optimization. In this paper, we introduced the notion of *persistent* TDG. This optimization was

[4]Default is $10,000,000$

[5]https://reviews.llvm.org/D63196

a major step to enable fine task grain depth-first scheduling, by reducing discovery costs by 15. We presented our implementation, and its impact on codes in Section 3.2, and believe it should be somehow adopted by production runtimes and standard specifications. Yet, refining tasks bellow 80μs on HPCG will likely require new advances in parallel runtime system, as studied in ParSEC [29].

MPI/OpenMP Interoperability. All our experiments are assuming the capacity of MPI and OpenMP to interoperate, *i.e.* automatically overlapping communication with computation from OpenMP task scheduler. This behavior is not granted by standards and many solutions exist [2–4, 10, 25, 26]. The most portable currently is the OpenMP `detach` clause approach [2, 10], but the MPI counterpart has not yet been standardized, and users must adapt their code to perform the interoperability 'by-hand'. We believe automatic runtime interoperability is a more suitable solution.

Porting Applications to task-based MPI + OpenMP. Implementing standard and efficient task-based versions of LULESH and HPCG with MPI+OpenMP was challenging. Fine synchronizations through dependent tasks is error prone, and task dependences-aware debugger, profiler and visualization tools are needed but none exists for the standard hybridation understudy. Porting LULESH from its `parallel for` version to dependent OpenMP tasks, the original user code was extended from 3,000 LoC and 45 `#pragma` to 4,500 LoC and 72 `#pragma` for task grain and dependency management.

6 RELATED WORK

Optimizations in task based runtime. The OpenMP dependent task model was adopted since the specifications 4.0 [30] but implementations induces important overhead [31]. The history of optimization in task graph runtime starts at least twenty years before OpenMP 4.0. Cilk [32] was the precursor by considering end-to-end optimizations from the compilation a fast and slow versions of each task to the implementation of the stealing protocol without costly lock operation on the general case execution path. This was the starting point to several paper related to work stealing scheduler optimization which is outside the scope of our TDG optimization. Parsec [33] based their graph model on the parametrized graph model [34] to implicitly represent tasks and their dependencies. It was a huge step in optimizing the memory space related to a task graph. Nevertheless, because this model does not fit well with irregular applications, the authors have also added a the classical task graph model proposed since the mid-90s [22, 23, 35, 36] built at run-time from description of tasks and their accesses (read,write) to the memory. Task graph optimization presented in section 3 are almost present in all these runtime, but our runtime is the first OpenMP runtime that systematically integrates them.

Persistent task graph. Moreover, we show that persistent task graph greatly reduces overhead in task graph discovery. Kaapi [35] is able to partition and distribute the data flow task graph to be reuse during iterative computation on distributed architecture. It was developed for checkpoint/recovery protocols. The overhead is high: it includes the task graph construction and the scheduling of the graph. [35] does not report any timings on representative HPC application. In [24] the authors proposed an extension of the OpenMP tasking model to be able to capture the graph unroll by a code section in order to re-execute it on next iteration: all the closures are captured during first execution included firstprivate data passed by value. Because only simulated results are reported with no runtime implementation and experimental results, it is difficult to compare and evaluate the performance of the proposition. [37] follows similar approach in capturing the tasks to replay the closure during iterative computation.

Unlike these approaches, our proposal tries to only cache internal data structure and allocation occurring at task creation. The producer re-executes the program instruction flow on every iterations: firstprivate can be updated for instance. Also, as opposed to [37], our approach allows the concurrent TDG discovery and execution. It provides *bounds* on total execution time and memory use through edges pruning or tasks throttling, for instance. Finally, we report evaluation of the cost of creating persistent task graph as well as the impact due to faster task graph discovery on the overall application performance.

7 CONCLUSION AND FUTURE WORK

The portable and efficient programming and execution of HPC scientific applications motivated the development of the two independent programming standards MPI and OpenMP. Recent task-based model of OpenMP provides new possible interactions with MPI.

In this paper, we showed that the OpenMP Task Dependency Graph (TDG) discovery has an important impact on computational and communication performances. We showed that combining fine task grains and a depth-first scheduler improves the use of the memory hierarchy and communications overlap. It lead us to accelerate the TDG discovery with runtime and code optimizations, implementing a new *persistent* TDG extension into MPC-OMP compliant with scientific simulation code needs. Our results showed up to 2.0x performances gain on LULESH weak-scaled to 16,000 cores against `parallel for` reference versions, and 1.2 speedup on state of the art task-based version.

For the future, we would like to see tasking *persistence* in OpenMP specifications, as it is a major step towards accelerating TDG discovery and reaching fine grain depth-first scheduling. Additionally, we believe the TDG discovery could have impacts on accelerators offloading, with similar effects onto SM memory and CPU/GPU communications, which we would like to investigate. Finally, more interactions between MPI and dependent task-based OpenMP are to be explored to improve programming and performances.

REFERENCES

[1] Sanjay Chatterjee, Sagnak Tasırlar, Zoran Budimlic, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. Integrating Asynchronous Task Parallelism with MPI. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 712–725, 2013.

[2] Joachim Protze, Marc-André Hermanns, Matthias S Müller, Van Man Nguyen, Julien Jaeger, Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. MPI detach - Towards automatic asynchronous local completion. *Parallel Computing*, 109:102859, March 2022.

[3] Kevin Sala, Xavier Teruel, Josep M. Perez, Antonio J. Peña, Vicenç Beltran, and Jesus Labarta. Integrating blocking and non-blocking MPI primitives with task-based programming models. *Parallel Computing*, 85:153–166, 2019.

[4] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. Argobots: A Lightweight Low-Level Threading and Tasking

Framework. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):512–526, 2018.

[5] Larry Meadows and Ken-ichi Ishikawa. OpenMP Tasking and MPI in a Lattice QCD Benchmark. In Bronis R. de Supinski, Stephen L. Olivier, Christian Terboven, Barbara M. Chapman, and Matthias S. Müller, editors, *Scaling OpenMP for Exascale Performance and Portability*, Cham, 2017. Springer International Publishing.

[6] Joseph Schuchart, Keisuke Tsugane, José Gracia, and Mitsuhisa Sato. The Impact of Taskyield on the Design of Tasks Communicating Through MPI. In *Evolving OpenMP for Evolving Architectures*. Springer International Publishing, 2018.

[7] Jérôme Richard, Guillaume Latu, Julien Bigot, and Thierry Gautier. Fine-Grained MPI+OpenMP Plasma Simulations: Communication Overlap with Dependent Tasks. In Ramin Yahyapour, editor, *Euro-Par 2019: Parallel Processing*, pages 419–433, Cham, 2019. Springer International Publishing.

[8] Manuel Ferat, Romain Pereira, Adrien Roussel, Patrick Carribault, Luiz-Angelo Steffenel, publisher="Springer International Publishing" editor="Klemm Michael Gautier, Thierry", Bronis R. de Supinski, Jannis Klinkenberg, and Brandon" Neth. Enhancing MPI+OpenMP Task Based Applications for Heterogeneous Architectures with GPU Support. In *OpenMP in a Modern World: From Multi-device Support to Meta Programming*, pages 3–16, Cham, 2022.

[9] Francesco Massimo, Mathieu Lobet, Julien Derouillat, Arnaud Beck, Guillaume Bouchard, Mickael Grech, Fréderic Pérez, and Tommaso Vinci. A Task Programming Implementation for the Particle in Cell Code Smilei. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '22, New York, NY, USA, 2022. Association for Computing Machinery.

[10] Joseph Schuchart, Philipp Samfass, Christoph Niethammer, José Gracia, and George Bosilca. Callback-based completion notification using MPI Continuations. *Parallel Computing*, 106:102793, 05 2021.

[11] Vinicius et al. Garcia Pinto. A Visual Performance Analysis Framework for Task-based Parallel Applications running on Hybrid Clusters. *Concurrency and Computation: Practice and Experience*, 30(18):1–31, April 2018.

[12] Elliott Slaughter, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S. Morris, Qinglei Cao, George Bosilca, Seema Mirchandaney, Wonchan Leek, Sean Treichlerk, Patrick McCormick, and Alex Aiken. Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.

[13] Ian Karlin. LULESH Programming Model and Performance Ports Overview. 2012.

[14] Ian Karlin, Jeff Keasler, and Neely Rob. LULESH 2.0 Updates and Changes. 2013.

[15] Guy E. Blelloch and Phillip B. Gibbons. Effectively sharing a cache among threads. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, page 235–244, New York, NY, USA, 2004. Association for Computing Machinery.

[16] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. Caliper: Performance Introspection for HPC Software Stacks. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 550–560, 2016.

[17] I Karlin, J McGraw, J Keasler, and B Still. Tuning the LULESH Mini-app for Current and Future Hardware. 2013.

[18] Marcos Maroñas, Xavier Teruel, and Vicenç Beltran. OpenMP Taskloop Dependences. In Simon McIntosh-Smith, Bronis R. de Supinski, and Jannis Klinkenberg, editors, *OpenMP: Enabling Massive Node-Level Parallelism*, pages 50–64, Cham, 2021. Springer International Publishing.

[19] Te Phhh, Shirley Moore, Jack Dongarra, N. Garner, K. London, and Phil Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14, 07 2000.

[20] Stephen L. Olivier, Bronis R. de Supinski, Martin Schulz, and Jan F. Prins. Characterizing and mitigating work time inflation in task parallel programs. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2012.

[21] Nathan R. Tallent and John M. Mellor-Crummey. Effective Performance Measurement and Analysis of Multithreaded Applications. *SIGPLAN Not.*, 44(4):229–240, feb 2009.

[22] François Galilée, Jean-Louis Roch, Gerson G. H. Cavalheiro, and Mathias Doreille. Athapascan-1: On-Line Building Data Flow Graph in a Parallel Language. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, page 88, USA, 1998. IEEE Computer Society.

[23] Alejandro Fernández, Vicenç Beltran, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. Task-Based Programming with OmpSs and Its Application. In Luís Lopes, Julius Žilinskas, Alexandru Costan, Roberto G. Cascella, Gabor Kecskemeti, Emmanuel Jeannot, Mario Cannataro, Laura Ricci, Siegfried Benkner, Salvador Petit, Vittorio Scarano, José Gracia, Sascha Hunold, Stephen L. Scott, Stefan Lankes, Christian Lengauer, Jesús Carretero, Jens Breitbart, and Michael Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops*, pages 601–612, Cham, 2014. Springer International Publishing.

[24] Chenle Yu, Sara Royuela, and Eduardo Quiñones. *Enhancing OpenMP Tasking Model: Performance and Portability*, pages 35–49. 09 2021.

[25] Seonmyeong Bak, Oscar Hernandez, Mark Gates, Piotr Luszczek, and Vivek Sarkar. Task-Graph Scheduling Extensions for Efficient Synchronization and Communication. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 88–101, New York, USA, 2021. Association for Computing Machinery.

[26] Romain Pereira, Adrien Roussel, Patrick Carribault, and Thierry Gautier. Communication-Aware Task Scheduling Strategy in Hybrid MPI+OpenMP Applications. In *17th International Workshop on OpenMP*, OpenMP : Enabling Massive Node-Level Parallelism, pages 1–15, Bristol, United Kingdom, September 2021.

[27] Jack Dongarra, Michael Heroux, and Piotr Luszczek. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *International Journal of High Performance Computing Applications*, 30, 08 2015.

[28] Alejandro Duran, Julita Corbalan, and Eduard Ayguade. An adaptive cut-off for task parallelism. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–11, 2008.

[29] J. Schuchart, P. Nookala, T. Herault, E. F. Valeev, and G. Bosilca. Pushing the Boundaries of Small Tasks: Scalable Low-Overhead Data-Flow Programming in TTG. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 117–128, Los Alamitos, CA, USA, sep 2022. IEEE Computer Society.

[30] Alejandro Duran, Josep M. Perez, Eduard Ayguadé, Rosa M. Badia, and Jesus Labarta. Extending the OpenMP Tasking Model to Allow Dependent Tasks. In Rudolf Eigenmann and Bronis R. de Supinski, editors, *OpenMP in a New Era of Parallelism*, pages 111–122, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[31] Thierry Gautier, Christian Pérez, and Jérôme Richard. On the Impact of OpenMP Task Granularity. In *14th International Workshop on OpenMP for Evolving Architectures*, pages 205–221, Barcelone, Spain, September 2018. Springer.

[32] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 212–223, New York, NY, USA, 1998. Association for Computing Machinery.

[33] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A Generic Distributed DAG Engine for High Performance Computing. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1151–1158, 2011.

[34] Michel Cosnard and Emmanuel Jeannot. Automatic Parallelization Techniques Based on Compact DAG Extraction and Symbolic Scheduling. *Parallel Processing Letters*, 11(1):151–168, 2001.

[35] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. *PASCO'07: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, pages 15–23, 07 2007.

[36] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[37] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. Cpp-Taskflow: Fast Task-Based Parallel Programming Using Modern C++. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 974–983, 2019.