Enhancing Load-Balancing of MPI Applications with Workshare

Thomas Dionisi[1], Stephane Bouhrour[1], Julien Jaeger[1,2,3](✉), Patrick Carribault[2,3], and Marc Pérache[2,3]

[1] Exascale Computing Research Laboratory, 2 Rue de la Piquetterie, 91680 Bruyères-le-châtel, France
[2] CEA, DAM, DIF, 91297 Arpajon, France
julien.jaeger@cea.fr
[3] Université Paris-Saclay, CEA, Laboratoire en Informatique Haute Performance pour le Calcul et la simulation, 91680 Bruyères-le-châtel, France

Abstract. Some high-performance parallel applications (e.g., simulation codes) are, by nature, prone to computational imbalance. With various elements, such as particles or multiple materials, evolving in a fixed space (with different boundary conditions), an MPI process can easily end up with more operations to perform than its neighbors. This computational imbalance causes performance loss. Load-balancing methods are used to limit such negative impacts. However, most load-balancing schemes rely on shared-memory models, and those handling MPI load-balancing use too much heavy machinery for efficient intra-node load-balancing. In this paper, we present the *MPI Workshare* concept. With *MPI Workshare*, we propose a programming interface based on directives, and the associated implementation, to leverage light MPI intra-node load-balancing. In this work, we focus on loop worksharing. The similarity of our directives with OpenMP ones makes our interface easy to understand and to use. We provide an implementation of both the runtime and compiler directive support. Experimental results on well-known mini-applications (MiniFE, LULESH) show that *MPI Workshare* succeeds in maintaining the same level of performance as well-balanced workloads even with high imbalance parameter values.

1 Introduction

The ever-growing need for computational power by simulation programs led to larger and more complex supercomputer architectures. Simulating natural phenomena can be complex and their evolving natures may lead to imbalanced workloads during execution. For example, considering a set of particles in a 3D space, they will be spread across all computing workers. If the initial distribution of the particles favored similar workloads on all computing workers, the balance may shift once the particles move through the 3D space and lead to some computing workers having to deal with more particles than their neighbors. Such

L. Sousa et al. (Eds.): Euro-Par 2021, LNCS 12820, pp. 466–481, 2021.
https://doi.org/10.1007/978-3-030-85665-6_29

workload imbalance causes performance loss. To circumvent this problem, different work balancing methods emerged, mainly relying on shared-memory capabilities. If mixing MPI with a shared-memory programming model (MPI+X) is now common, hence providing a favorable ground for load-balancing, such hybrid programming is not always ideal regarding performances. First, runtime stacking, i.e., having the runtimes for each programming model running at the same time, is a challenge. Each runtime, often developed as if it will be the only runtime running with the application, might make decisions that would hurt the performance of the other running runtime. Second, the load-balancing will only happen within the scope of the shared-memory models. This means that even if multiple MPI processes are on the same node, no computational load-balancing will ever happen. MPI load-balancing has been proposed in the past, where message passing concepts are frequently used to support both inter-node and intra-node load balancing. Although this methodology is necessary for inter-node, using a distributed-memory mechanism can be heavy to handle intra-node load-balancing.

This paper exposes the *MPI Workshare* concept, which aims at providing the simplicity of OpenMP worksharing constructs to the MPI scope. We propose a set of directives, inspired by OpenMP, with a work stealing runtime system to provide shared-memory load-balancing to legacy MPI codes. Using *pragmas* similar to those of OpenMP, we ensure an easy-to-adopt interface for users, with a minimal impact on the program. In this first implementation, we focus on loop worksharing. The contributions of this paper are: 1) the presentation of the *MPI Workshare* concept, 2) the definition of the *MPI Workshare* programming interface, easy-to-adopt and leveraging incremental evolution of MPI (and MPI+OpenMP) codes to include MPI shared-memory constructs, 3) an implementation of directive compiler support and *MPI Workshare* runtime for loop worksharing, and 4) performance evaluations of MPI load-balancing on intra-node and inter-node benchmarks (pure MPI and MPI+OpenMP).

The paper is organized as follows. Section 2 documents related work to highlight the lack of efficient intra-node MPI load-balancing. The *MPI Workshare* concept is presented in Sect. 3, along with details on the loop workshare focus, and definition of the *MPI Workshare* interface. Section 4 focuses on the runtime and implementation details. Then experimental results are displayed in Sect. 5, before concluding in Sect. 6.

2 Related Work

Work-sharing concepts have long been studied in HPC. The first version of the OpenMP standard exposes such constructs through the `pragma omp for` and `pragma omp sections` directives. The loop worksharing construct distributes the associated loop iterations to all OpenMP threads, either statically or dynamically. Thus OpenMP have seen numerous work trying to improve the balance of its worksharing. [5] introduced a novel loop scheduling option, named *adaptive*. The scheduling strategy is based on the static scheduler by creating a per-worker queue. They enable a work stealing scheduler between the workers to dynamically balance the work along the execution when a thread becomes idle. Contrary to [18] where the number of stolen tasks is statically defined, a worker is allowed to steal the half of its victim's queue. In [19], the authors show that a state-of-the art runtime loop schedule is not efficient enough, and a mixed-approach with polyhedral compiler analysis driving the runtime decision can leverage much better performances. Recently, [2] exposes the state of the art of OpenMP loop scheduling and argues for the need of more scheduling policies. The introduction of tasking in the OpenMP standard led to studies aiming to improve the scheduling of numerous task [3,11,12,20]. These studies mainly advocates for local work-stealing to avoid extra costs.

Work sharing principles have already been integrated in some MPI applications. Most of these studies rely on an MPI+X approach, with the load-balancing enabled for intra-node only through a thread-based programming model [16]. Few studies looked at real MPI load-balancing. In [15], authors use a Divide-and-Conquer algorithm based on MPI dynamic processes to improve performance of an N-Queens problem. FLEX-MPI [9] provides a whole library on top of MPICH-2, to dynamically evaluate the load imbalance with hardware counters and online MPI profiling, and redistributes the data through communications. However, the user needs to register all data use in the computation for the runtime to automatically load-balance the work, which is very cumbersome and can induce some overheads. A more recent approach uses fine grain integration of tasking with MPI [8], using Task-Aware MPI [17]. If this work allows to have better scheduling of task including MPI communications, and tasks depending on these communications, the load-balancing is still confined to the scope of one MPI process. Most papers really targeting MPI load-balancing rely on message passing to exchange redistributed data. Though it is mandatory for inter-node load balancing, intra-node load balancing can be lighter by relying on shared-memory principles directly in the MPI runtime.

For several years, Partitioned Global Address Space (PGAS) programming models extend the shared memory paradigm across a whole cluster. This *unified* view of the memory allows embedding load-balancing concepts more easily in their runtimes. The authors of [4] implements an optimized (lock queue reduction, stealing more than one tasks at a time, ...) work stealing scheduler thanks to the PGAS *ARMCI*, able to scale on a large distributed system. The HabaneroUPC++ [6,7] PGAS model focus on work stealing scheduler thanks to the concatenation of the Habanero task programming model and the UPC++ PGAS tool. In [10], the authors introduce a dynamic tasking library for UPC, with a new Hierarchical Victim Selection (HVS) method to preserve locality. The main problem of PGAS models is the necessity to rewrite the program with their semantics.

Intra-node inter-process load-balancing is either too code intrusive with PGAS, or too heavy-weight with regular MPI load-balancing. However, a recent work presents simple intra-node *communication* load-balancing [13]. So why the same kind of concept is not also applied to intra-node computation load-balancing? Our *MPI Workshare* concept aims at leveraging the simplicity and efficiency of OpenMP worksharing, but between MPI processes on the same node. The proposed programming interface inspired by OpenMP ensures a quick understanding of its use, and limits the efforts and impacts on the source code, compared to a PGAS oriented rewrite. As we focus on intra-node MPI load-balancing, this work is complementary to all inter-node load-balancing schemes, and also with shared-memory model load-balancing such as OpenMP worksharing constructs.

3 MPI Workshare

MPI Workshare offers worksharing features to MPI, to leverage some load balancing in one of the most used parallel programming API. This section first presents the general idea of *MPI Workshare*, before describing our loop workshare implementation and the proposed interface.

3.1 *MPI Workshare* Concept

MPI Workshare exposes to users a way to enable inter-MPI process load-balancing on some specific parts of the program. Thus it is composed of two parts:

1. a programming interface allowing users to identify the parts of the local work that will be exposed to other MPI processes, and
2. a runtime implementation handling these parts and leveraging a stealing mechanism between MPI processes.

Marking the code parts that will be exposed to other MPI processes is left to the user. To ease the adoption of our *MPI Workshare* interface, this step requires to be the less invasive possible in the code, and easy to grasp and to use. To this end, we propose a programming interface inspired by OpenMP and its worksharing mechanism. The code selection for *MPI Workshare* is done through pragmas, with a set of `#pragma ws` directives inspired from OpenMP ones. These directives are detailed in Sect. 3.2. Based on these directives, the *MPI Workshare* runtime will transform the selected work into subtasks that can be executed by other MPI processes.

The second part is the work stealing mechanism, and when to actually steal work from another MPI process. In an MPI program, most performance loss comes from the synchronization induced by the MPI communications. Stealing work means that the local MPI process will have to check is there is some work to steal from another MPI process. This verification induces communications, hence synchronization, which can be harmful to the global performances. To avoid adding extra synchronization, probing for work to steal is done only when the local MPI process is already in a waiting state due to the MPI semantics. Instead of just waiting for the pending communications, the MPI runtime activates the *MPI Workshare* runtime to check if it can help any busy MPI process. With this behavior, the original semantics of the MPI process is kept, and no additional synchronizations are inserted due to the *MPI Workshare*.

Loop Workshare. In this work, we limit the scope of the *MPI Workshare* approach to *for* loops. The method used for the *MPI Workshare* on loops is very similar to the one used in OpenMP for the loop workshare constructs (e.g., `#pragma omp for` directive). Thus the loop iterations are decomposed into chunks. One major difference with the OpenMP construct is that these chunks are not initially spread onto the available MPI processes. Each MPI process is still in charge of executing its own loop, as expected from a usual MPI program. However, these chunks are exposed to the other MPI processes for workshare. If another MPI process finishes its work and ends up in a waiting state, it can steal available chunks. Figure 1 displays this behavior with 3 MPI processes located on the same node. Rank 0 has no loop, whereas ranks 1 and 2 have each one for loop, symbolized with the plain lines. Rank 1 has less iterations than rank 2. All MPI processes synchronize through an MPI call, represented with the dotted lines. With no workshare (left frame), rank 0 just waits for rank 1 and rank 2 to finish their work. Rank 2 has the heavier load, and delays the completion time of all ranks. Thanks to workshare (right frame), rank 0 starts to look for chunks to steal as soon as it enters the synchronizing call. It first steals rank 1, helping to finish its local work sooner. Then, both ranks are in the synchronizing call, and they both steal chunks from rank 2. Thanks to the stealing, rank 2 finishes its local work more quickly. It enters the synchronizing call earlier, thus "freeing" the other ranks from the synchronizing call earlier too.

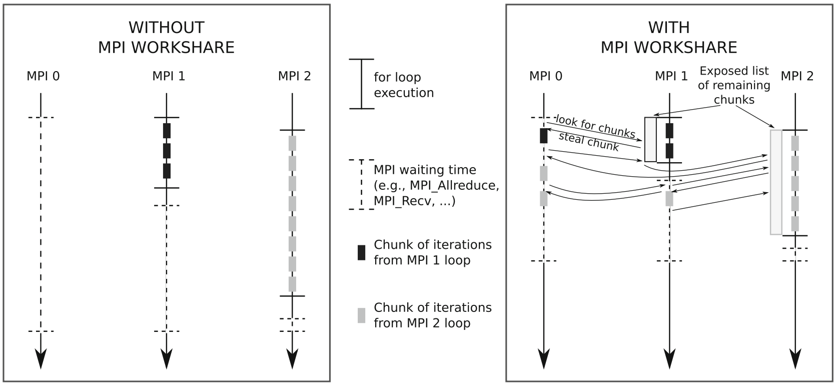
WITHOUT MPI WORKSHARE
MPI 0
MPI 1
MPI 2
for loop execution
MPI waiting time (e.g., MPI_Allreduce, MPI_Recv, ...)
Chunk of iterations from MPI 1 loop
Chunk of iterations from MPI 2 loop
WITH MPI WORKSHARE
MPI 0
MPI 1
Exposed list of remaining chunks
MPI 2
look for chunks
steal chunk

Fig. 1. *MPI Workshare* mechanism on 3 MPI processes, 2 MPI processes having a for loop with a `#pragma ws for` directive.

Like for OpenMP, it is possible as well to provide additional information to influence the sizes and numbers of chunks. It is also possible to define the equivalent of the `guided` and `dynamic` clauses of the `#pragma omp for` directive to have different size of chunks for the same loop, to leverage finer load balancing. All the new directives provided for *MPI Workshare* are detailed in the following Section.

Workshare Workers for Spare Cores. In regular MPI+OpenMP programs, there are less MPI processes than compute resources, as to leave compute resources dedicated to the OpenMP threads that will be spawned during execution. When *MPI workshare* is applied on a loop, it prevents OpenMP to be applied on the same loop. This means: 1) iterations won't be distributed among OpenMP threads and will execute sequentially (from the OpenMP perspective) and, 2) compute resources dedicated to OpenMP threads may remain idle as the corresponding OpenMP threads won't be awake. To avoid this, the *MPI Workshare* spawns *workers* on the idle resources. Each MPI process has its own set of *workers* on its local idle resources. Each *worker* helps performing local chunks (current MPI process) and work stealing (remote MPI process). With this policy, the *workers* behave for the local MPI processes in a fashion similar to OpenMP threads with a `#pragma omp for` directive. It is possible for a compute resource to host both an OpenMP thread and a workshare *worker*. However, they won't be active at the same time.

3.2 *MPI Workshare* Interface

The *MPI Workshare* interface is based on 3 directives, inspired by OpenMP standard, that enable loop worksharing.

Directive `#pragma ws for`. The main directive is `#pragma ws for`.

```
#pragma ws for [clause[ [,] clause] ... ] new-line
loop-nest
```

where loop-nest is a canonical loop nest and clause is one of the following:

private(list)

firstprivate(list)

lastprivate([lastprivate-modifier:]list)

reduction([reduction-modifier,]reduction-identifier:list)

schedule([modifier [, modifier]:]kind[, chunk_size])

collapse(n)

steal_schedule([modifier [, modifier]:]kind[, chunk_size])

This directive acts as the `#pragma omp for` in the OpenMP specification, but in the MPI scope. This construct specifies that the iterations of the associated loop/loopnest will be exposed to other MPI processes. Thus if an MPI process is in a waiting phase and is allowed to steal work, it will be able to execute some of the exposed iterations in parallel of the local MPI process. If multiple MPI processes are in this position, then the iterations will be distributed across all stealing MPI processes.

Both the `schedule` and `steal_schedule` clauses specify how iterations of the associated loop(s) are divided into chunks, and how these chunks are distributed among the MPI processes. The `schedule` clause drives the chunk choice only on the local MPI process. The `steal_schedule` clause specifies how other MPI processes will steal loop iterations. Both of these clauses accepts the same values as the `schedule` clause in the OpenMP specification: static, dynamic or guided.

All other clauses that can be passed to the `#pragma ws for` directive are similar to those of the `#pragma omp for` construct in the OpenMP specification. For example, it is possible to include a reduction operation in the scope of the `#pragma ws for` directive through the use of `reduction` clause with the right operator and the final variable.

Directive `#pragma ws atomic`. As the `reduction` clause cannot cover all the *reducing* operations that can happen in a loop body, the OpenMP specification offers the possibility to specify atomic instructions through the `#pragma omp atomic` construct. To offer the same level of expressiveness, we also provide the directive `#pragma ws atomic`.

```
#pragma ws atomic [clause[ [,] clause] ... ] new-line
statement
```

For this directive, the clauses, and according statements, supported are the same as for the `#pragma omp atomic` in the OpenMP specification. The behavior is also similar, as it ensures that the specified storage location is atomically updated, to avoid race conditions due to concurrent accesses of multiple MPI processes.

Directive `#pragma ws critical`. The `#pragma ws critical` is a generalization of the `#pragma ws atomic`. The construct applies on a scope defined with a structured block.

```
#pragma ws critical [(name)]  new-line
structured-block
```

This construct ensures that the code in the scope will be executed only by one MPI process at a time. The accepted clauses are similar to the ones accepted by the `#pragma omp critical` in the OpenMP specification.

4 Implementation

We implemented *MPI Workshare* into the MPC runtime [14]. MPC provides both an OpenMP and an MPI implementation, with the MPI runtime having both process-based and thread-based flavors. In the latter, all MPI processes in a node are in fact threads in one encompassing OS process. In this mode, the *MPI Workshare* runtime in MPC is very similar to the OpenMP runtime. Inside a node, all MPI processes are threads able to access the same memory space. An *MPI Workshare* structure is created for each MPI process. To help other

MPI ranks, an MPI process iterates on the *MPI Workshare* structure of each other MPI process until finding one with unfinished exposed work. Such work is symbolized by a *shared index* inside the *MPI Workshare* structure with a value between the lower and upper bound of the shared loop. To steal a chunk, the MPI process has to get the current iteration chunk index, and update its value to the next chunk. Thus this MPI process will be in charge of performing the loop iterations in the selected chunk. These three operations are performed atomically with a *compare_and_swap* operation. State-of-the-art stealing optimizations have been implemented (stealing from the end of the list) along with several victim selection policies (MPI process with the most iterations remaining, or with less thieves, or closest according to hardware topology). The whole stealing method is realized with a lock-free implementation.

MPI Calls in Shared Loops. It is possible that the loop tagged for workshare contains MPI calls. Thus an MPI process stealing iterations should act as if it was indeed the former MPI process performing the call. The MPC runtime provides such features through the `MPIX_Disguise` function [1]. It allows an MPI process to temporarily assume the identity of another MPI process in the same compute nodes. Thanks to this feature, our implementation enables workshare even with MPI calls in the associated loop body.

However, we have the following restrictions for the moment. First, it is forbidden to access/modify the locations of data in the loop with RMA procedures. Second, we apply the same restriction on the loop iterations than OpenMP: there must be no semantic dependencies between iterations (either data dependencies, or ordering dependencies such as with MPI collective initialization procedures). The last restriction could be leveraged. OpenMP proposed the `ordered` directive to enforce that part of the parallel loop should be executed in order according to the iterations. As future plan, we envision to implement such `ordered` directive for *MPI Workshare*. With such addition, it can then be possible to balance loops containing MPI collective initialization procedures, as long as they are protected with the `ordered` directive.

Compilation and OpenMP Compatibility. Directives require compiler support to translate the *MPI Workshare* pragmas into the associated runtime calls. We implement this translation into GCC 7.3.0 (the most-recent supported by MPC at the time the work was performed) for C and C++, based on the existing OpenMP pass. We provide a new flag `-fws`, that enables *MPI workshare* directives. It is possible to have both OpenMP and *MPI workshare* directives in one program, and to use both `-fws` and `-fopenmp` flags. The runtimes are compatible and can work concurrently. However it is not possible to have both OpenMP and *MPI workshare* constructs on the same loop.

5 Experimental Results

To evaluate our *MPI Workshare* concept, we realized several experimentations on a platform composed of dual-socket Intel Xeon Platinum 8168 (Skylake) nodes, each with 24 cores at 2.7 GHz, equipped with Mellanox MT27700 (Connect-IB) InfiniBand boards. Results are grouped into two categories: pure MPI runs, and hybrid MPI+OpenMP executions. For pure MPI runs, workshare results are produced without any *workers*, as all compute resources are populated with MPI processes. For MPI+OpenMP runs, *MPI Workshare* spawns *workers* on each compute resources originally used by OpenMP.

All results with *MPI Workshare* are realized using MPC 3.4.0. We compare those results to both MPC 3.4.0 without activating *MPI Workshare* and openmpi 2.0.4. When OpenMP is involved, openmpi uses GCC 7.3.0 OpenMP runtime, and MPC 3.4.0 without *MPI Workshare* uses its own OpenMP runtime relying on GCC 7.3.0 for directive translation. For such tests, MPC 3.4.0 with *MPI Workshare* replaces OpenMP loop pragmas with *MPI Workshare* pragmas, and OpenMP threads are replaced with workshare *workers*.

5.1 Pure MPI Benchmarks with *MPI Workshare*

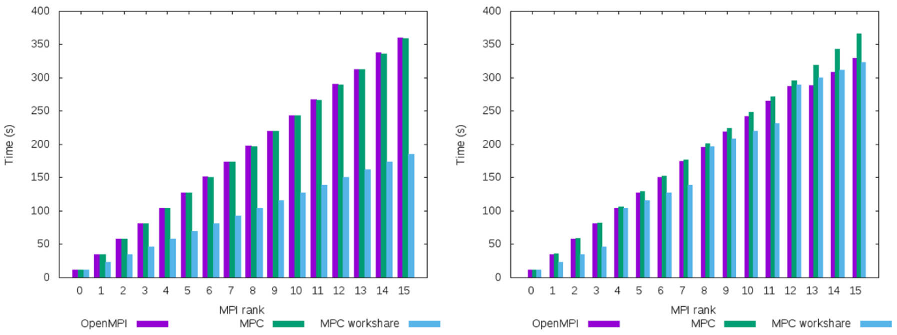

Fig. 2. Timing of each MPI ranks on a microbenchmark with imbalanced workload (see Listing 1.1), without (`openmpi`, `MPC`) and with *MPI Workshare*. Left: 16 MPI processes on a single node. Right: 16 MPI processes on 4 nodes (4 MPI processes per node). (Color figure online)

Double-Loop Microbenchmark. Listing 1.1 shows an example of imbalanced microbenchmark based on nested loops Iterations of the outer loop have monotonically increasing workload. Thus, the iterations of this loop are distributed across multiple MPI processes, to expose imbalanced workload across MPI.

Figure 2 (left) displays the execution time for 16 MPI processes located on the same node. The first two bars are for openmpi and MPC runs without workshare. The timings convey the monotonically increasing workload on each MPI process. The last bar (blue) is for MPC with *MPI Workshare*. The timings reported are

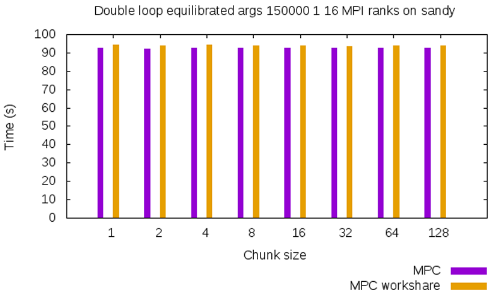

Fig. 3. *MPI Workshare* overhead evaluation on balanced workload, on a single node using 16 cores.

for the local completion of the loop nest on each rank. The timing of the whole benchmark is the same for each rank, and is similar to the local completion time of the slowest rank, due to `MPI_Finalize` synchronizing effect. As described in Sect. 3.1, *MPI Workshare* runtime is active only when the MPI runtime is idle. Thus, the *MPI Workshare* activates at the call to `MPI_Finalize` after the loop nest. The difference between the local timing and the global timing (local rank vs slowest rank) is the time spent waiting in the `MPI_Finalize` call, hence the time the MPI process can help other MPI processes by stealing loop iterations.

Rank 0 is the rank with the least workload. It performs all its iterations without help, hence its local completion time is the same with workshare (blue bar) than without (purple and green bars). Once its work is done, it waits in the barrier, and start looking for iterations to steal. Rank 1 is the first to be stolen, with a local completion timing lower with workshare. One after another, all waiting ranks start stealing the following MPI processes. Ultimately, rank 15, which is the rank with the greater workload, reaches the barrier nearly twice as fast as regular MPI execution, thanks to *MPI Workshare*.

Figure 2 (Left) is the best case for *MPI Workshare*, with all MPI processes on the same compute node. Figure 2 (Right) displays the results of executing the same benchmark on 4 nodes with 4 MPI processes per node. Once again, *MPI Workshare* allows the blue bar to grow slower than usual MPI. This is because *MPI Workshare* works only on intranode. Hence, workload is balanced only between the 4 MPI processes on the same node.

Overhead Study on Balanced Workload. Using the same microbenchmark based on nested loops, we change the inner loop so each iteration will perform the same computation as the other iterations, completely balancing the workload. We kept the same global number of iterations over all the MPI processes. This configuration allows to evaluate the overhead of *MPI Workshare* on a balanced workload. The results are displayed in Fig. 3. We varied the size of the chunk of iterations (1 meaning 1 iteration per chunk, 128 meaning 128 iterations per chunk). As we can see, on a balanced workload, the performance using

MPI Workshare is very close to the pure MPI approach. The observed overhead remains under 2% of the total execution time.

Lulesh and MiniFE Miniapps. Lulesh and miniFE (from the CORAL suite on resp. hydrodynamic and finite elements) both offer a parameter to insert workload imbalance between MPI processes. We ran Lulesh on an Intel KNL processor because of Lulesh restrictions. Indeed, Lulesh can only run with a number of MPI processes which is a cube. We opted for 64 MPI processes, and the KNL architecture was the only one available to us which allowed to have all 64 MPI processes on the same node. MiniFE experimentations are performed on the Skylake platform. Both experimentations display similar performances in intra-node (Fig. 4 for Lulesh and Fig. 5 (Left) for miniFE). For openmpi or standard MPC, the performance progressively drops when increasing the load imbalance parameter. However, with *MPI Workshare*, the performance stays constant regardless of the load imbalance.

```
MPI_Init();
...
#pragma ws loop
for i = MPIid*N/P to (MPIid+1)*N/P
{
  for j = 0 to i
  {
    do_some_work();
  }
}
...
MPI_Finalize();
```

Source Code 1.1: Imbalance loop workload on MPI processes

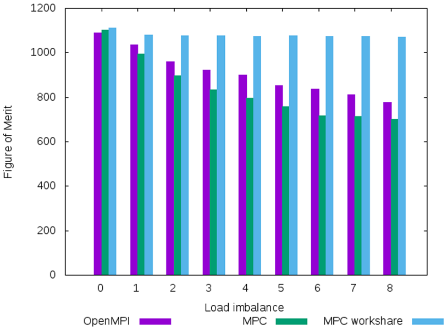

Fig. 4. Performance on Lulesh against the load imbalance with size = 100 and 64 MPI processes on KNL processors

Figure 5 (Right) displays miniFE behavior for varying number of nodes, with a load imbalance parameter fixed to 500. We observe that *MPI Workshare* still leverages better performance than usual MPC runs, though the speed-up being less important with a greater number of nodes, due to the intra-node scope limiting the workshare.

5.2 MPI+OpenMP Benchmarks with *MPI Workshare Workers*

We also evaluated *MPI Workshare* on MPI+OpenMP applications by replacing some OpenMP loop directives by the *Workshare* counterpart. *MPI Workshare* will spawn *workers* to populate the compute resources left vacant by the removed OpenMP directive. Hence, in intra-node, all the 48 cores are always used with a fitting combination of MPI processes with either OpenMP threads or *workers* (e.g., 1 MPI process with 48 threads, or 8 MPI processes with 6 threads).

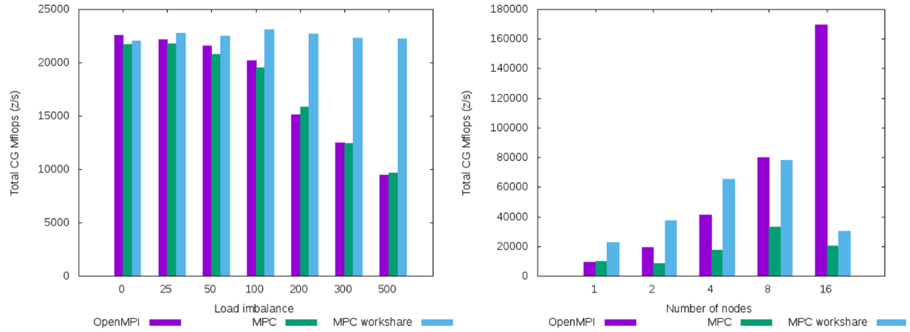

Fig. 5. Performance on miniFE with nx = ny = nz = 200 and 48 MPI processes per node on Skylake processors. Left: results on one node, x-axis is the percentage of imbalance as proposed by miniFE configuration. Right: results for varying number of nodes with a fixed load-imbalance percentage of 500.

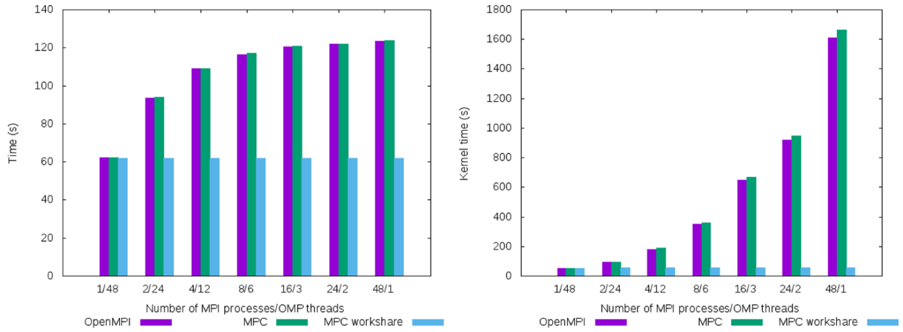

Fig. 6. Comparing MPI+OpenMP with MPI+*MPI Workshare* with *workers*. Second number of x-axis is number of OpenMP threads for `openmpi` and `MPC`, and number of *MPI Workshare workers* for `MPC+Workshare`. Left: microbenchmark with imbalanced workload (see Listing 1.1). Right: GAP Triangle Counting (TC) benchmark.

Double-Loop and GAP Triangle Count Microbenchmarks. We adapted microbenchmark Listing 1.1 and the Triangle Count benchmark from GAP Suite to run MPI+OpenMP tests. The Triangle Count benchmark is very highly imbalanced parallelized with OpenMP, relying on dynamic scheduling to enable load balancing. Since it does not use MPI, we did a basic MPI implementation distributing the same number of iterations between the MPI processes (similar to our double-loop benchmark). In our microbenchmark, we added OpenMP directives on the outer loop. Thus, for both benchmarks, when running with openmpi and MPC, the loop is parallelized with OpenMP in each MPI process, and for MPC with *MPI Workshare*, loop iterations are exposed to other MPI processes. Both benchmarks display similar behaviors (see Fig. 6). For 1 MPI process on the node, results are similar. With both OpenMP and *MPI Workshare*, the whole workload is balanced on the 48 availables cores. For more than 1 MPI process per node, OpenMP cannot balance the whole workload, as its scope is limited to intra-MPI process. With *MPI Workshare*, the whole workload continues to be perfectly balanced between all the *workers* of all MPI processes, keeping the same performance for every configuration.

miniFE miniapp. Figure 7 (Left) exposes results running MPI+OpenMP version of miniFE (MPI being either openmpi or MPC) compared to MPC+*MPI Workshare* using *workers* with different combinations of MPI processes and threads. Pure OpenMP executions of the miniapp showed that with a number of threads higher than 8, OpenMP scalability is degrading and performances collapse. This can be observed on the MPI+OpenMP version. Up to 4 MPI processes, the number of OpenMP threads are higher than 8, and the performances are driven by the OpenMP poor scalability. Hence, since performance issues do not come from the MPI workload imbalance, *MPI Workshare* do not bring any improvement. However, with more than 4 MPI processes, the OpenMP scalability is good, and the performance is driven by the MPI imbalance. Thus, the *MPI Workshare* brings better performance for such configurations.

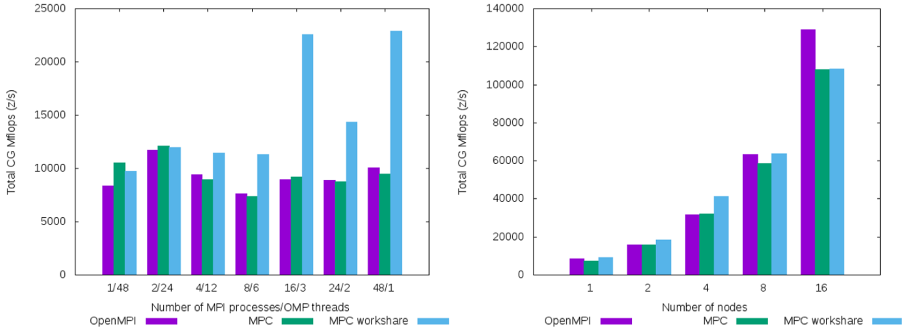

Fig. 7. MiniFE performance with an imbalance of 500. Second number of x-axis is number of OpenMP threads for `openmpi` and `MPC`, and number of *MPI Workshare workers* for `MPC+Workshare`. Left: varying the number of MPI processes, and accompanying threads/*workers*, on one node. Right: varying number of nodes, fixed number 8 MPI ranks with 6 threads/*workers* per node.

Figure 7 (Right) displays the internode performance with 8 MPI processes and 6 OpenMP threads, or 6 workshare *workers*. For a large number of nodes, the intra-node scope of *MPI Workshare* prevents it from having much benefit. However, for a small number of nodes, *MPI Workshare* succeeds in improving the performance.

6 Conclusion

MPI load-balancing can be critical to leverage good performance. Though most load-balancing mechanism are available on intra-node for other programming models, MPI load-balancing existing techniques rely on the heavy machinery of message passing. In this paper, we presented the *MPI Workshare* concept, offering the possibility to annotate an MPI program source code to allow inter-MPI processes intra-node load-balancing. The proposed interface is inspired from OpenMP directives to facilitate its adoption. We described our implementation of *MPI Workshare* directives and runtime, targeting loop iteration worksharing.

We tested our implementation on several microbenchmarks and CORAL miniapps. We showed that intra-node load-balancing is very efficient even with heavy load imbalance for pure MPI runs. In inter-node, *MPI Workshare* also provides speed-up, though it is inherently limited by its intra-node scope. With MPI+OpenMP applications, the addition of workshare *workers*, to populate the cores occupied by idle OpenMP threads, enables combining intra-MPI process iterations distribution with inter-MPI process load-balancing.

The *MPI Workshare* concept shows promising results on the tested benchmarks. However, our implementation only applies on regular loops, hence limiting the scope in actual simulation programs. In future work, we aim to extend *MPI Workshare* to other worksharing constructs inspired by OpenMP, such as `sections` and `tasks`. These constructs would allow a user to apply *MPI Workshare* load-balancing to independent parts of the code outside of regular for loops, which is mandatory to be efficient on real-life applications.

Acknowledgments. This work was performed under the Exascale Computing Research collaboration, with the support of CEA and UVSQ.

References

1. Besnard, J.B., et al.: Mixing ranks, tasks, progress and nonblocking collectives. In: Proceedings of the 26th European MPI Users' Group Meeting. EuroMPI 2019, Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3343211.3343221
2. Ciorba, F.M., Iwainsky, C., Buder, P.: OpenMP loop scheduling revisited: making a case for more schedules. In: de Supinski, B.R., Valero-Lara, P., Martorell, X., Mateo Bellido, S., Labarta, J. (eds.) IWOMP 2018. LNCS, vol. 11128, pp. 21–36. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98521-3_2
3. Clet-Ortega, J., Carribault, P., Pérache, M.: Evaluation of OpenMP task scheduling algorithms for large NUMA architectures. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014. LNCS, vol. 8632, pp. 596–607. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09873-9_50
4. Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable work stealing. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–11 (November 2009). https://doi.org/10.1145/1654059.1654113
5. Durand, M., Broquedis, F., Gautier, T., Raffin, B.: An efficient OpenMP loop scheduler for irregular applications on large-scale NUMA machines. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 141–155. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40698-0_11

6. Kumar, V., Murthy, K., Sarkar, V., Zheng, Y.: Optimized distributed work-stealing. In: 2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3), pp. 74–77 (November 2016). https://doi.org/10.1109/IA3.2016.019
7. Kumar, V., Zheng, Y., Cavé, V., Budimlić, Z., Sarkar, V.: Habaneroupc++: A compiler-free pgas library. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014, pp. 5:1–5:10. ACM, New York (2014). https://doi.org/10.1145/2676870.2676879
8. Maroas, M., Teruel, X., Bull, J.M., Ayguad, E., Beltran, V.: Evaluating worksharing tasks on distributed environments. In: 2020 IEEE International Conference on Cluster Computing (CLUSTER), pp. 69–80 (2020). https://doi.org/10.1109/CLUSTER49012.2020.00017
9. Martín, G., Marinescu, M.-C., Singh, D.E., Carretero, J.: FLEX-MPI: an MPI extension for supporting dynamic load balancing on heterogeneous non-dedicated systems. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 138–149. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40047-6_16
10. Min, S.J., Iancu, C., Yelick, K.: Hierarchical work stealing on manycore clusters. In: 5th Conference on Partitioned Global Address Space Programming Models, p. 35 (2011)
11. Muddukrishna, A., Jonsson, P.A., Vlassov, V., Brorsson, M.: Locality-aware task scheduling and data distribution on NUMA systems. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 156–170. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40698-0_12
12. 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
13. Ouyang, K., Si, M., Hori, A., Chen, Z., Balaji, P.: Cab-mpi: Exploring interprocess work-stealing towards balanced mpi communication. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, IEEE Press (2020)
14. Pérache, M., Carribault, P., Jourdren, H.: MPC-MPI: an MPI implementation reducing the overall memory consumption. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) EuroPVM/MPI 2009. LNCS, vol. 5759, pp. 94–103. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03770-2_16
15. Pezzi, G.P., Cera, M.C., Mathias, E., Maillard, N., Navaux, P.O.A.: On-line scheduling of mpi-2 programs with hierarchical work stealing. In: 19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2007), pp. 247–254 (October 2007). https://doi.org/10.1109/SBAC-PAD.2007.36
16. Ravichandran, K., Lee, S., Pande, S.: Work stealing for Multi-core HPC clusters. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011. LNCS, vol. 6852, pp. 205–217. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23400-2_20
17. Sala, K., et al.: Improving the interoperability between mpi and task-based programming models. In: Proceedings of the 25th European MPI Users' Group Meeting, EuroMPI 2018, Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3236367.3236382
18. Subramaniam, S., Eager, D.L.: Affinity scheduling of unbalanced workloads. In: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing, pp. 214–226. IEEE Computer Society Press, Los Alamitos, CA, USA (1994). http://dl.acm.org/citation.cfm?id=602770.602810

19. Thoman, P., Jordan, H., Pellegrini, S., Fahringer, T.: Automatic OpenMP loop scheduling: a combined compiler and runtime approach. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 88–101. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30961-8_7
20. Virouleau, P., Broquedis, F., Gautier, T., Rastello, F.: Using data dependencies to improve task-based scheduling strategies on NUMA architectures. In: Dutot, P.-F., Trystram, D. (eds.) Euro-Par 2016. LNCS, vol. 9833, pp. 531–544. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43659-3_39