Evaluating Worksharing Tasks on Distributed Environments

1st Marcos Maroñas
Barcelona Supercomputing Center (BSC)
mmaronas@bsc.es

2nd Xavier Teruel
Barcelona Supercomputing Center (BSC)
xteruel@bsc.es

3rd J. Mark Bull
EPCC, University of Edinburgh
m.bull@epcc.ed.ac.uk

4th Eduard Ayguadé
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
eayguade@bsc.es

5th Vicenç Beltran
Barcelona Supercomputing Center (BSC)
vbeltran@bsc.es

***Abstract*—Hybrid programming is a promising approach to exploit clusters of multicore systems. Our focus is on the combination of MPI and tasking. This hybrid approach combines the low-latency and high throughput of MPI with the flexibility of tasking models and their inherent ability to handle load imbalance. However, combining tasking with standard MPI implementations can be a challenge. The Task-Aware MPI library (TAMPI) eases the development of applications combining tasking with MPI. TAMPI enables developers to overlap computation and communication phases by relying on the tasking data-flow execution model. Using this approach, the original computation that was distributed in many different MPI ranks is grouped together in fewer MPI ranks, and split into several tasks per rank. Nevertheless, programmers must be careful with task granularity. Too fine-grained tasks introduce too much overhead, while too coarse-grained tasks lead to lack of parallelism. An adequate granularity may not always exist, especially in distributed environments where the same amount of work is distributed among many more cores. Worksharing tasks are a special kind of tasks, recently proposed, that internally leverage worksharing techniques. By doing so, a single worksharing task may run in several cores concurrently. Nonetheless, the task management costs remain the same than a regular task. In this work, we study the combination of worksharing tasks and TAMPI on distributed environments using two well known mini-apps: HPCCG and LULESH. Our results show significant improvements using worksharing tasks compared to regular tasks, and to other state-of-the-art alternatives such as OpenMP worksharing.**

I. Introduction

Hybrid programming combines two different programming models to exploit inter-node and intra-node parallelism. This is a promising approach to exploit modern HPC systems, but effectively combining intra-node and inter-node parallelism remains a challenging task, because it usually requires a seamless integration of two different approaches in the same application.

This paper focuses on the combination of message passing (MPI) and tasking to exploit inter-node and intra-node parallelism, respectively. Our goal is to mix the low latency and high throughput of MPI together with the flexible data-flow execution model of tasks, and also to leverage the inherent ability of tasks to overlap computation and communication phases, as well as to handle intra-node load imbalance.

We focus on tasking rather than other approaches because of its ability to drive the execution based on the data-flow, and the minimal effort required to add tasking to an already written application. Also, tasking enables an incremental parallelization. Other approaches would require deeper changes or even a complete refactor of applications.

However, such a combination present several challenges. The use of MPI communications inside tasks can easily lead to deadlocks. Additionally, communication overhead can increase considerably due to the fact that tasks can be blocked waiting for communications to complete. If this happens, all the threads blocked will be idle while other tasks may be ready to run, and are waiting for some thread to execute them. The Task-Aware MPI library [1] is a library that simplifies the interoperability between MPI and tasking models [2]. The use of TAMPI guarantees the progress of MPI operations inside tasks, ensuring a deadlock-free execution. Additionally, with TAMPI, the programmer can leverage fine-grained synchronization between tasks to overlap communication and computation phases. This is achieved by releasing the cores of tasks blocked on communications to make them available to run other ready tasks.

In this hybrid approach, the computations in MPI ranks that belong to the same node are grouped together into a few ranks (usually one per NUMA node). The work on each of these few ranks is then split into multiple tasks. This parallelization strategy introduces another challenge related to the task granularity. Finding the proper task granularity is a concern in shared-memory environments [3] but its importance might be even higher in distributed environments. Too fine-grained tasks result in too much overhead, while too coarse-grained tasks result in lack of parallelism. In distributed environments, given that the number of cores is usually much higher compared to shared-memory environments, it is easy to incur a lack of parallelism, especially in strong scaling scenarios.

The OmpSs-2 programming model [4] recently presented

978-1-7281-6677-3/20/$31.00 ©2020 IEEE

a new type of task called *worksharing task* [5]. This is a special kind of task that can be applied to `for` loops to uncover additional parallelism. Worksharing tasks efficiently exploit the fine-grained structured parallelism of `for` loops dividing the iteration space into chunks, that can be executed collaboratively by several cores, but creating only a single task. Thus, the overhead is kept low while parallelism is increased.

In this work, we study the combination of worksharing tasks and TAMPI in weak and strong scaling scenarios using two well-known mini-apps such as LULESH and HPCCG. Specifically, our contributions are (1) completely taskified and distributed versions of LULESH and HPCCG, (2) versions of LULESH and HPCCG using worksharing tasks and MPI, (3) a thorough analysis of the different versions, and (4) an evaluation of the weak and strong scaling scenarios using a large number of cores. Our results show significant improvements: for LULESH reaching up to 1.3x speedup compared to the best state-of-the-art version (pure MPI), and 3x compared to regular tasks; and for HPCCG reaching up to 1.08x speedup compared to the best state-of-the-art version (OpenMP+MPI), and 1.33x speedup compared to regular tasks.

The rest of this document is structured as follows: Section II contains background to understand this work; Section III details the proposed solution; Section IV consists of an evaluation and discussion of the proposal; Section V reviews the most relevant related work; Section VI summarizes the work done and provides concluding remarks; and, finally, Section VII presents future work proposals.

II. Background

In this Section we describe the worksharing tasks and the TAMPI library to make our proposal comprehensible.

Worksharing tasks are a special kind of task that behaves like a regular task in almost everything, but there is a key difference that makes them very useful: regular tasks are run by a single thread, while worksharing tasks may be run collaboratively by several threads concurrently, while still keeping the task management overhead as low as for a regular task. In terms of OpenMP [6], it can be compared to a task with a `parallel for` inside, but with no barrier at the end of the construct. Thus, parallelism is increased, while the overhead remains unchanged.

Worksharing tasks are only applicable to `for` loops. The iteration space of the loop is partitioned into chunks and each of the chunks can run in a different thread at the same time. In consequence, the iterations of the loop must be independent. A worksharing task will use up to as many threads as cores available, at the point where the task is scheduled for execution.

In summary, worksharing tasks preserve the main advantage of using tasks (i.e. a data-flow execution with its consequent flexibility), while keeping overhead low, and enabling sufficient parallelism.

TAMPI is a library designed to improve the interoperability between task-based programming models and MPI. Placing MPI calls inside tasks may cause deadlocks due to the out-of-order execution of tasks. This library implements a cooperation mechanism between the tasking runtime and MPI library that ensures a deadlock-free and efficient execution.

TAMPI provides two different modes: blocking and non-blocking. In this paper, we only use the non-blocking mode. This mode focuses on the use of non-blocking asynchronous MPI operations inside tasks. When using TAMPI non-blocking mode inside a task, the task binds its completion not only to the execution of its body, but also to the completion of all the MPI requests indicated. The task completion implies the release of its dependences, as well as freeing its data structures.

TAMPI offers two methods to describe which requests a task must wait for, both of them asynchronous and non-blocking: `TAMPI_Iwait` for a single request, and `TAMPI_Iwaitall` for multiple requests. So, a task that binds its completion to one or more MPI requests using the mentioned methods will not complete (and so release its dependences) until its body is run *and* the MPI requests have completed. If a task completes its body before one or more MPI requests finished, it will not release its dependences, and so, the successors cannot become ready for execution. However, the core that was running the unfinished task can proceed to execute other ready tasks, thus preventing the core to be idle waiting for the communication.

When the communication actually finishes, TAMPI notifies the task-based runtime system that the task is actually completed. After that, the runtime can release the dependences, and the successors become ready for execution. With this mechanism, progress is ensured. The effects are not only deadlock-free executions, but there are also possible improvements in performance, because the CPU utilization tends to be better.

In normal MPI programs, the programmer can try to overlap communication and computation by placing non-blocking communication calls as early as possible and the corresponding waits as late as possible, and hope that the MPI library progresses the communication while the intervening computation is being executed. By using tasks and TAMPI, the MPI calls do not require such careful placing, available computation can be discovered dynamically via the task dependency system, and reliance on the MPI progress engine is reduced.

III. Using Worksharing Tasks (+TAMPI) in Distributed Environments

In Section I we stated the problems that we face when combining tasking and message passing. In this section we give an in-depth explanation of these problems. In addition, we will discuss why the use of worksharing tasks and TAMPI can solve, or minimize the negative impact of, such problems.

Task granularity is a serious concern inherent to the use of tasks. The lower bound is determined by the runtime ability to manage fine-grained tasks, while the upper bound is determined by a function of the problem size and the number of cores. Either of the edges is extremely harmful for performance: at the lower edge there are too many fine-grained tasks that introduce too much overhead, while at the

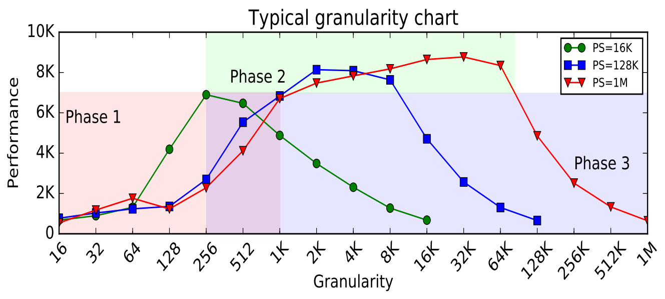

Fig. 1. Typical granularity chart.

upper edge there are too few coarse-grained tasks that cannot feed all the cores.

Figure 1 shows such a problem. In the figure, the x-axis shows granularity, and the y-axis shows performance. There are three different series representing different problem sizes. The chart also contains coloured parts which represent different chart (not application) phases. Phase 1 shows the scenario where many fine-grained tasks introduce too much overhead, Phase 2 shows the scenario reaching peak performance, and Phase 3 shows the scenario where there are too few coarse-grained tasks that cannot feed all the cores. Note that from x=256 to x=1K, phases 1 and 3 are overlapped. This is because it is Phase 1 for PS=16K, but Phase 3 for PS=128K and PS=1M.

Problem size is a further factor to consider in the granularity choice: the bigger the problem size is, the coarser grain we can use without incurring in lack of parallelism. This fact can also be seen in Figure 1. All of this is true in shared-memory environments. However, a distributed environment aggravates the task granularity problem. The reason is that we usually try to solve the same problem faster by providing more resources. Therefore, in a strong-scaling scenario, the overall problem size remains constant, but it must be distributed amongst an increasing number of cores. If we keep increasing the number of cores while the problem size and the task granularity remains the same, the performance will suffer due to lack of parallelism. The problem size cannot be changed in this scenario, because it is the problem we want to solve, so we can only adjust the task granularity.

Of course, we can reduce the task granularity as we increase the number of cores. Nonetheless, we will end up in the other edge: too many fine-grained tasks that introduce too much overhead. It is exactly at this point where worksharing tasks can be a useful tool. Using worksharing tasks, a single task can feed several cores (up to all the cores of a single MPI process), but with an overhead similar to a regular task, which can only feed a single core. Thus, coarser granularities can still be used without incurring either a lack of parallelism or too much overhead.

Communication overhead can be another important problem when combining tasking and message passing. Even if we use asynchronous MPI operations, there is always a point where the data is actually needed. If data is not ready yet, given that we include the communication operations inside the tasks, they will have to wait for the data. This happens because the task-based runtime system has no clue about whether a task may need to wait for communications or not. For instance, suppose there are two different tasks ready to run: task A needs data from communication and task B does not. The runtime chooses task A and data from the communication is not ready yet, so the core is just waiting idle until the data is ready. Meanwhile, task B could have been completed. If this situation is sufficiently frequent, performance can be negatively affected.

TAMPI prevents the previous situation from happening. Properly annotating code with data dependences, and using the appropriate TAMPI methods, a task that needs data from a communication will not start until the data is available because the dependences will prevent it from occurring. Thus, in the previous situation, task B would have been executed first, and then task A after the predecessors completed (i.e. the communications finished). Overall, the CPU utilization improves because the cores are no longer idle waiting for communications to complete, but can run other ready tasks instead. Compared to not using TAMPI, performance may improve if cores were idle waiting for communications to complete for sufficiently long.

IV. Analysis and Evaluation of Applications

Our evaluation uses two well known mini-apps: **High Performance Computing Conjugate Gradients (HPCCG)** [7] and **Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)** [8]. Both of them are well known proxy apps used to assess the performance of software and hardware.

HPCCG is a simple conjugate gradient benchmark code for a 3D chimney domain. It is part of the Mantevo project [9]. This mini-app presents very well structured parallelism, and it is memory-bound.

LULESH solves the Sedov blast wave problem for one material in three dimensions [10]. It is one of the five challenge problems in the DARPA UHPC program. It is designed to test different tuning techniques and programming models. LULESH contains different phases and includes load imbalance issues.

The reference versions for both apps are the pure MPI and the OpenMP+MPI versions available in the public repositories of each of them. Starting from these versions, we developed two new versions using regular tasks and worksharing tasks. We would like to clarify that the version using regular tasks uses OmpSs-2, as does the worksharing tasks version. Also, both of them make use of TAMPI, while the OpenMP+MPI and the pure MPI versions do not.

Next we will detail the methodology followed in the evaluation. After that, we present, analyze and discuss the results from the two mini-apps.

A. Methodology

As already introduced, our evaluation involves four different versions. The names that appear in the legends of figures as follows:

- **MPI**. The version using only MPI.
- **OMP**. The version using OpenMP worksharing loops and MPI.
- **T**. The version using regular tasks of OmpSs-2 and MPI. It includes TAMPI.
- **TF**. The version using worksharing tasks of OmpSs-2 and MPI. It includes TAMPI.

For each of the applications, we perform a granularity analysis in a single NUMA node. This is because in the strong and weak scaling experiments we use one MPI process per NUMA node. The objective is to determine the best granularity to be used in the weak and strong scaling scenarios.

It may happen that a single granularity does not fit all the tasks. We support our analysis with execution traces. In the traces, we show the weights of each task so that it is possible to see if granularities are adequate for some tasks but not for others. When this happens, we define a reference granularity and then we apply multiplication factors for those tasks that need it. For instance, suppose there are two tasks: task A and task B. The granularity is 10 iterations. This granularity is good for task A, but it is too fine for task B and we spend as much time creating the task as executing it. Therefore, we apply a multiplication factor of 2 for task B, so its granularity is now 20 iterations. The points of the granularity analysis will always refer to the reference granularity.

In addition, we motivate the decision of using worksharing tasks rather than regular tasks using execution traces. For obvious reasons, task granularity analysis does not apply to the `MPI` version. The `OMP` version uses only worksharing loops with static scheduling, so the granularity analysis does not affect it either.

Once the optimal granularity is determined, we move forward to our second analysis: scalability analysis in a distributed environment. This analysis incorporates two scenarios: weak scaling and strong scaling. Weak scaling starts from 1 node and a given problem size. At each new point, the number of nodes is increased and so is the problem size. Strong scaling also starts from 1 node and a given problem size. However, in this scenario, the problem size is fixed for all the points, while the number of nodes is increased at each new point. As a last remark, in this experiment, we use one MPI rank per core for the pure MPI version, and one MPI rank per NUMA node (to favour data locality) for the hybrid versions.

Regarding the execution environment, all the experiments were carried out on Marenostrum 4. A node of Marenostrum 4 is composed of 2 sockets Intel Xeon Platinum 8160 2.1GHz 24-core and 96GB of main memory [11]. Regarding the software, we used the Mercurium compiler [12] (v2.3.0), the Nanos6 runtime [13] (2020-05-15), the gcc and gfortran compilers (v7.2.0), and the Intel compilers (v17.0.4). Regarding MPI, we used the Intel implementation (v17.0.4). We also use the Intel implementation of OpenMP. Finally, we used Extrae [14] (v3.7.1) to obtain the execution traces, and Paraver [15] to visualize them.

Finally, we would like to highlight that each of the results is an average of 5 executions. We did not observe any significant variation between different executions (standard deviation <5%), so we think 5 executions is sufficient.

B. LULESH

LULESH is a quite big and complex mini-app. It contains two different main phases, one devoted to perform operations over elements, and the other one devoted to perform operations over nodes. There are several kernels, with very different computational costs. This application has a high degree of parallelism across all the execution, if correctly annotated with data dependences. Also, by design, it presents load imbalance problems. These two facts, that can be seen in Figure 2, make LULESH a very suitable application for tasking.

Figure 2 shows an execution trace of LULESH. The execution trace shows what is being executed in each of the threads (y-axis) over time (x-axis). The white color means that no task is being executed, and each of the other colors represent a different task type. In this trace, we show a single iteration of the main loop: from the red tasks to the next red tasks, which are already part of the next iteration. The cyan tasks are the last tasks belonging to the first phase, and the purple tasks mark the start of the second phase. The arrows on the top of the trace show approximately the duration of each phase.

When using MPI, there are four type of communications in the application. The position of each of the them is approximately indicated in Figure 2. At the beginning of an iteration, there is a *MPI_Allreduce*. Then there are two more point-to-point communications before the small orange tasks, and after the cyan tasks. Finally, the fourth and last communication, also point-to-point, can be found in the white space between the green and the dark orange tasks.

The OpenMP+MPI implementation uses worksharing loops (i.e. `parallel for`). Usually, there is a `parallel for` per loop. In some cases, the developers use the `nowait` clause that eliminates the implicit barrier at the end of a worksharing loop. As a side effect, this means that static scheduling is mandatory. Given that LULESH presents load imbalance issues, the use of worksharing with static scheduling may hinder performance. In contrast, tasks has an inherent ability to deal with load imbalance.

Moving from OpenMP worksharing to tasks requires replacing each of the worksharing loops by several tasks. The most important consideration is to avoid the use of `taskwait`, and rely on the data dependences to achieve lightweight synchronization. Once we have the implementation with tasks using data dependences, we need to determine how many different granularities there should be in the mini-app. For that purpose, we use execution traces.

The trace of Figure 2 is an execution using a large problem size, which displays good behavior throughout. Figure 3 is the same trace but using a smaller problem size. In this case, to have the same number of tasks, they must be more fine-grained. In fact, they are too fine-grained, and the task management overhead becomes too much. Specifically, in Figure 3 it is possible to see the white color dominating the trace, meaning that most of the time the cores are idle.

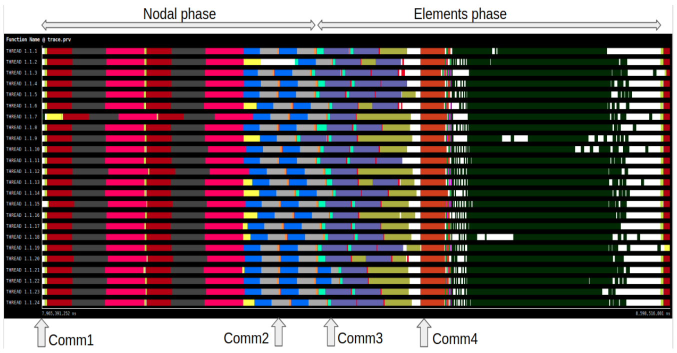

Fig. 2. Execution trace of the LULESH `T` version using a single granularity for all the tasks with a big problem size on 24 cores (1 NUMA socket)

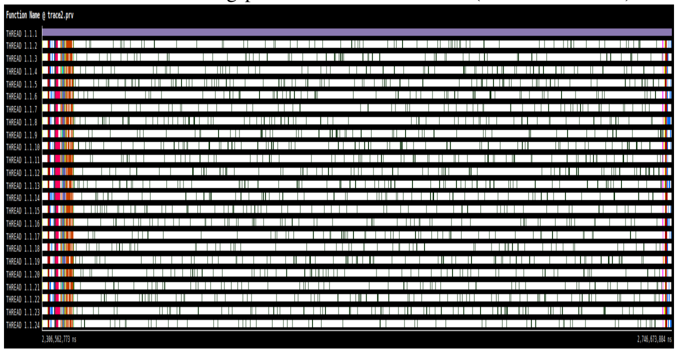

Fig. 3. Execution trace of the LULESH `T` version using a single granularity for all the tasks with a small problem size on 24 cores (1 NUMA socket)

Looking at the complete trace, it is possible to see the producer core (core 0, purple color) creating tasks during the entire execution. The consumers cores execute tasks faster than the producer core can create them.

The scenario with a small problem size is important considering the strong scaling experiment. Given that we set a fixed problem size and then we increase the number of processors, at each new step the problem size per process will decrease, ending up in scenarios such as the one in Figure 3 or with even smaller problem sizes.

One possible solution to this problem is using different granularities for different task types. Not all the task types have the same weight, even with the same granularity. Thus, for tasks with low weights, it may be better to use bigger granularities. Around a hundred microseconds is the minimum time we consider a task must last to be worth paying the management costs. Table I presents the average time a task of a given type requires to complete. In the table, each of the tasks types has a small square with the same color the task has in the execution trace of Figure 3. There are several task types whose average time that does not reach this threshold. Thus, we should change the granularity of several task types.

We increased the granularity of each task type to reach the 100 microsecond threshold. It is important to highlight that the increment of the granularity is not always directly proportional to the increment of time. That gave us the multiplication factors shown in Table I for each of the task types.

When we implement this, there is a ∼12x speedup in the execution time. Figure 4 shows how an iteration looks like using the listed factors. It is possible to see that the duration

TABLE I
Average task time, and multiplication factor, for each of the task types of LULESH mini-app using a small problem size

Task type	Average	Factor
CalcLagrangeElements 🟧	22.25 us	16
CalcVelocityAndPositionForNodes 🟩	46.07 us	4
check_eosv_vc 🟩	13.10 us	16
InitStressTermsForElems 🟨	13.98 us	24
UpdateVolumeForElems 🟪	5.12 us	48
CalcHourglassControlForElems ⬛	361.32 us	1
CalcTimeConstraintsForElems 🟨	81.99 us	2
CalcFBHourglassForceForElems 🟥	615.50 us	1
EvalEOSForElems 🟩	24.98 us	16
CalcFBH_collect ⬜	175.45 us	1
CalcMonotonicQRegionForElems 🟧	16.50 us	48
IntegrateStressTermsForElems 🟥	386.66 us	1
CalcMonotonicQGradientsForElems 🟨	212.95 us	1
IntegrateStressTermsForElems_collect 🟦	93.06 us	2
CalcKinematicsForElems 🟪	402.48 us	1
ApplyAccelerationBoundaryConditionsForNodes 🟥	3.86 us	96
CalcAccelerationForNodes 🟧	21.69 us	16

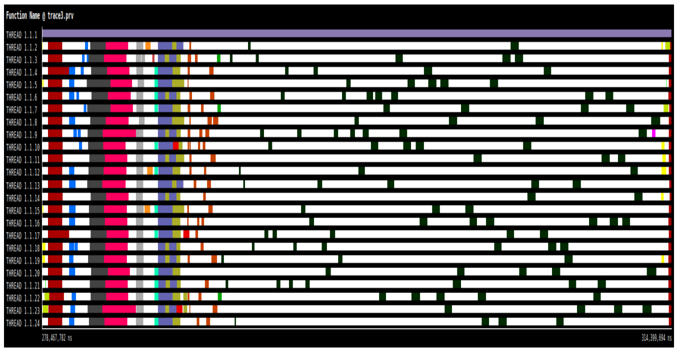

Fig. 4. Execution trace of the LULESH `T` version using multiple granularities for different task types with a small problem size on 24 cores (1 NUMA socket)

of the iteration is ∼440 ms in Figure 3 and ∼36 ms in Figure 4. Even so, there is still a lot of white color, meaning cores are running no tasks. In some regions of the execution trace this may be caused by dependences: tasks are waiting for their predecessors to finish so they can start. However, there are other regions where we know for sure there are no dependences between tasks, for instance in the big region with dark green tasks. Thus, we conclude that the problem is that the thread creating tasks is not creating them fast enough. In other words, we are creating too many tasks, so again we need to increase the granularity.

Figure 5 shows the granularity chart using a problem size of 50 elements per dimension. In the x-axis we show the number of created tasks/worksharing tasks per each of the different task types. The y-axis shows the figure of merit (FOM) of the mini-app. The `T` version reaches the peak performance with a reference granularity of 48 tasks per type, and the `TF` version reaches the peak performance using a reference granularity of 24 worksharing tasks per type. Considering thata NUMA node of the machine has 24 cores, there is 1 worksharing task per core in the `TF` version, and 2 tasks per core in the `T` version. A significant performance difference exists between using regular tasks and worksharing tasks. Regarding the implementation, the difference between the two versions is using

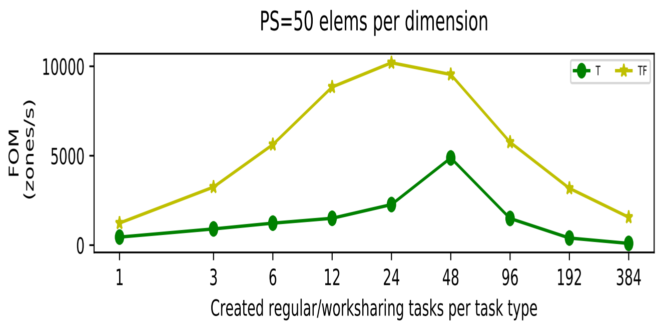

Fig. 5. Granularity chart of LULESH

a single worksharing task, rather than multiple tasks, in several portions of code. All the methods listed previously, where we apply a multiplication factor, use a single worksharing task, except *CalcVelocityAndPositionForNodes*, *CalcLagrangeElements*, *InitStressTermsForElems* and *CalcMonotonicQRegionForElems*. With such a simple change, the peak performance shows a speedup of ∼2.1x.

The main reason for such a big impact in performance is the significant reduction in terms of the number of created tasks. Fewer tasks means fewer overheads, and, usually, less parallelism. Notwithstanding, thanks to the use of worksharing tasks, creating fewer tasks does not affect parallelism. The number of created tasks is reduced to such an extent that the main thread is able to create enough work for ten iterations in the time taken to execute five iterations. In other words, for the regular tasks version, the main thread keeps creating tasks during the whole execution, while using worksharing tasks it requires only ∼50% of the execution time. Additionally, this means that the core devoted completely to creating tasks in the regular tasks version, can contribute to running some of the tasks in the worksharing tasks version.

Figure 6 shows an iteration of the TF version. In this case, the granularity used is the optimal one based on Figure 5. We can see there that an iteration takes ∼12 ms to complete. Even so, there are some problems that should be addressed. In particular, there are some regions suffering a lack of parallelism and others suffering load imbalance. Considering that the granularity used causes the creation of only 1 task per type per core, and that the introduction of multiplication factors reduces the number of created tasks, this makes sense. We can see that all the regions suffering a lack of parallelism or load imbalance are using regular tasks. Thus, a logical step forward is to replace those regular tasks by worksharing tasks. By doing so, each of the worksharing tasks can be executed by several cores, thus preventing lack of parallelism. Also, given that worksharing tasks allow threads to move forward when there is no remaining work, load imbalance should be also improved.

As a consequence of replacing all the regular tasks by worksharing tasks, it is likely that the optimal granularity is lower than the one used previously. Thus, we repeated the granularity analysis keeping the same problem size (50 elements per dimension), which is shown in Figure 7. The

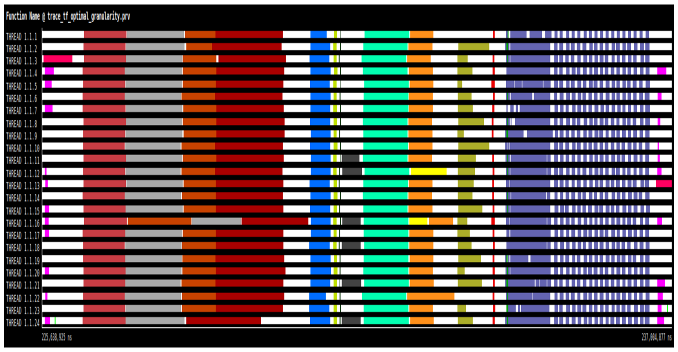

Fig. 6. Execution trace of the LULESH TF version using multiple granularities for different task types with a small problem size on 24 cores (1 NUMA socket)

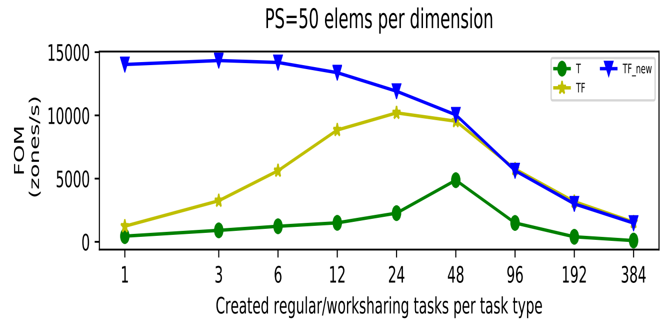

Fig. 7. Updated granularity chart of LULESH

peak performance in this new version occurs when using a reference granularity of three worksharing tasks per task type. This is because the fewer tasks created, the fewer overheads are introduced. Worksharing tasks enables us to create a lower amount of tasks per task type, while keeping enough work to feed all the cores.

Figure 8 shows an iteration of the updated TF version (i.e. using worksharing tasks wherever possible). The trace was obtained using the optimal granularity: 3 worksharing task per task type. Now an iteration only takes ∼9 ms to complete. So, to sum up, we started with a version where an iteration required ∼440 ms to complete, and ended up with a version where an iteration requires only ∼9 ms to complete.

After detailing the modifications done in the implementation, and selecting an adequate granularity, we are able to move forward and perform the experiments in a distributed

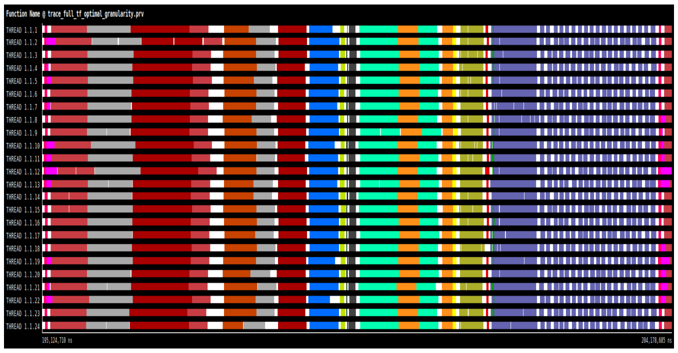

Fig. 8. Execution trace of the updated LULESH TF version (i.e. using worksharing tasks wherever possible) using multiple granularities for different task types on 24 cores (1 NUMA socket)

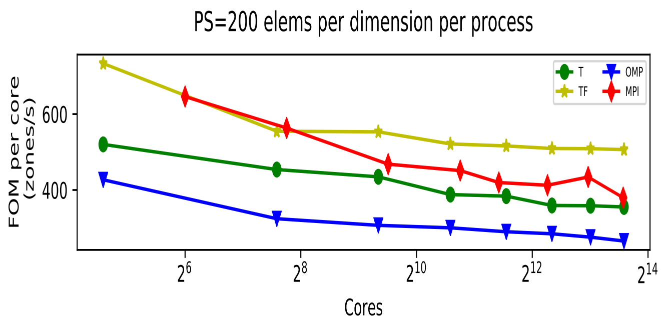

Fig. 9. Results of the LULESH weak scaling experiment

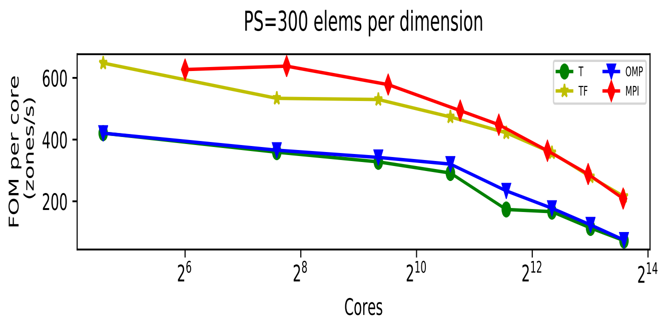

Fig. 10. Results of the LULESH strong scaling experiment

environment. Figure 9 shows the results of the weak scaling experiment, and Figure 10 shows the results of the strong scaling experiment. This application restricts the number of MPI processes used to be a cube of an integer number. That introduces difficulties in the comparison between the MPI version and the rest because the total number of cores used does not match. For this reason, we decided to use the FOM/core instead of the raw FOM metric.

Figure 9 shows in the x-axis the number of total cores used and in the y-axis the FOM per core of the application. For this experiment, we used a problem size of 200 elements per dimension per process. The OMP version is the worst across all the different scenarios, while the TF is the best across all the different scenarios. The MPI version begins close to the TF version but then it falls. Regarding the T version, it is in between the OMP and the MPI version until the last point where it obtains almost the same performance than the MPI version. Overall TF version is able to reach speedups of up to ∼1.4x, ∼1.9x, and ∼1.3x compared to the T, OMP, and MPI versions, respectively.

Figure 10 shows in the x-axis the number of total cores used and in the y-axis the FOM per core of the application. For this experiment, we used a problem size of 300 elements per dimension. The TF and the MPI versions perform very similarly across all the scenarios. Similarly, the T and the OMP versions behave very much alike. However, there is a significant difference between the two groups, reaching up to ∼2.8x speedup.

The reason for the performance improvement of the TF version compared to the T version is that the number of tasks is drastically reduced. Consequently, there is a drastic reduction of overhead. In addition, by creating fewer tasks, the creator

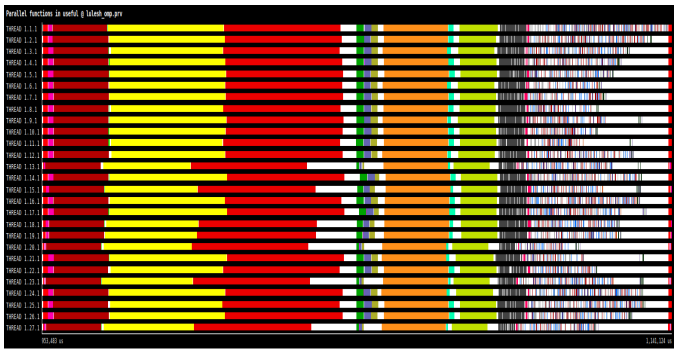

Fig. 11. Execution trace of the LULESH MPI version showing the time performing computations

can finish the creation earlier. Therefore, the cores rarely have to wait because the creator cannot create fast enough. Finally, this fact also means that the creator can start running tasks after finishing the creation.

Compared to the OMP version, the tasking versions introduces two key advantages: (1) the flexibility given by the data-flow execution model, and (2) the overlapping of computation and communication phases given by the TAMPI library. The OMP version is implemented using worksharing. At the end of each worksharing loop there is a barrier, where all the threads must wait until all of them have finished. This rigidness in the synchronization may introduce a significant performance penalty, especially if there is load imbalance, which is the case here. Also, the communication in the OMP version is always done outside parallel regions. Thus, when the data is required, threads in this version are idle while waiting for the communication to complete. In contrast, the T and the TF versions are able to keep progressing running other ready tasks.

Compared to the MPI version, the tasking versions introduce the key advantage of the overlapping of computation and communication phases. Figures 11 and 12 show the time performing computations and communications respectively. These figures evidence the amount of time that MPI version wastes in communications. In contrast, the tasking versions can keep progressing thanks to TAMPI. However, for the T version this is not enough due to the task management overheads. The TF version is able to reduce these overheads, and so, is able to outperform the MPI version consistently in the weak scaling experiment, reaching up to ∼1.3x speedup in the scenario with more cores. Regarding the strong scaling experiment, the TF version is competitive with the pure MPI version, and even obtains a slight speedup of 1.03x.

C. HPCCG

HPCCG is a simpler application compared to LULESH. It contains a single phase relying on three different kernels: ddot, waxpby and sparseMV. This application is very well suited for OpenMP worksharing loops, given its fork-join pattern shown in Figure 13. For each iteration, there is a ddot kernel that can run in parallel. After that, it computes the residual and the alpha, using at most two cores. This closes the parallelism because alpha is required by the following

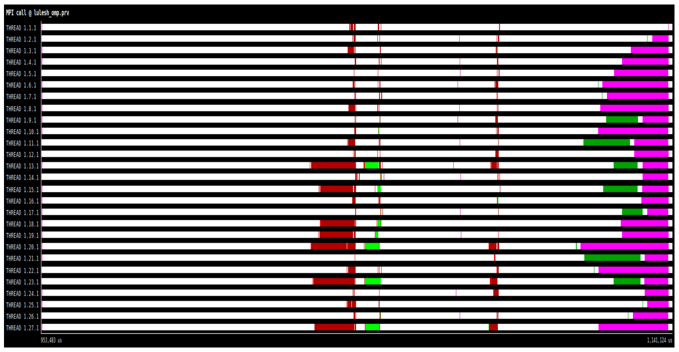
Fig. 12. Execution trace of the LULESH MPI version showing the time performing communications

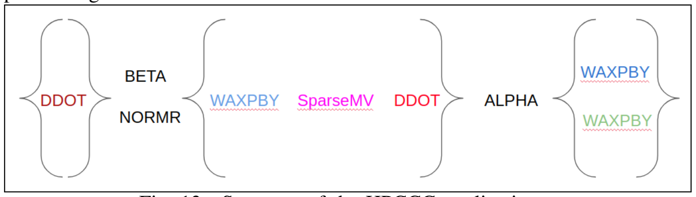

Fig. 13. Structure of the HPCCG application

tasks. Then, parallelism is open again to run the `waxpby`, `sparseMV`, and `ddot` kernels. Following this comes the compute of beta, which closes parallelism again because the following tasks need beta. Finally, the `waxpby` kernels can run in parallel again.

When using MPI, there are three different communications. There is an `MPI_Allreduce` after each of the two ddot kernels, and one point to point communication before the `sparseMV`. The result of the reductions is required to compute beta, normr, and alpha. Thus, the overlapping of computation and communication is only possible in the point to point communication.

This structure is not well suited for tasks because it forces all the cores to wait twice per iteration, as if there was a taskwait. As a consequence, there is not much benefit from using data dependences because there are implicit barriers imposed by the application structure.

The reference OpenMP+MPI implementation simply uses a `parallel for` in each of the three kernels. In the tasking version we replaced each of the `parallel for` by a set of tasks with the required data dependences. Even in an application with a fork-join pattern is important to avoid the use of taskwaits. A taskwait implies that no more tasks will be created until all the already created tasks finish. If we use data dependences, the tasks are already created, and so, as soon as the data is ready, they can run. By using taskwait, when the data is ready (all the previous tasks finished), it starts creating tasks (one by one) again, increasing the overall waiting time.

We again start our analysis by determining how many different granularities there should be in the mini-app. For that purpose, we use the execution trace shown in Figure 14. The execution trace shows what is being executed in each of the threads (y-axis) over time (x-axis). The white color means no task is being executed and each of the other colors represent a different task type. In this trace, we show a single iteration of the main loop: from the blue and green tasks to the next blue

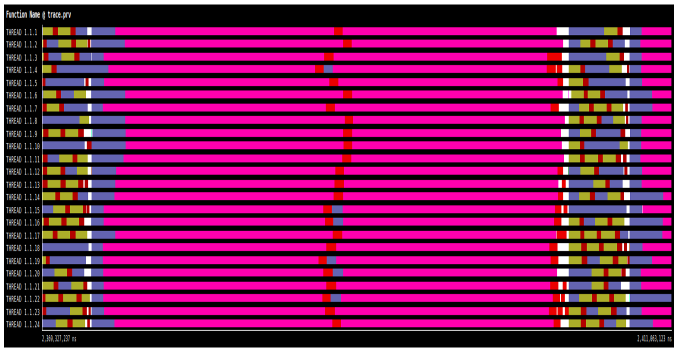
Fig. 14. Execution trace of the HPCCG T version using a single granularity for all the tasks with a big problem size on 24 cores (1 NUMA socket)

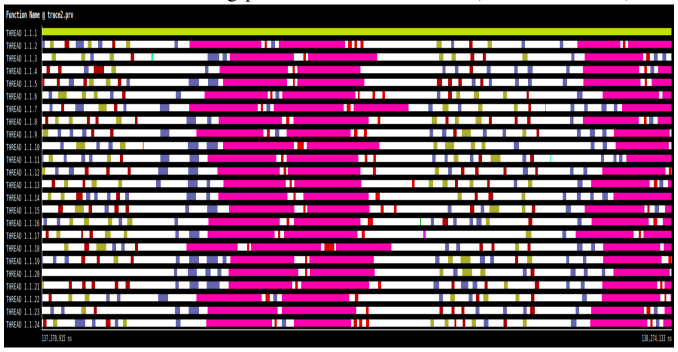
Fig. 15. Execution trace of the HPCCG T version using a single granularity for all the tasks with a small problem size on 24 cores (1 NUMA socket)

and green tasks. Actually, in this application, the iterations are overlapped. The green tasks and some of the blue tasks of the beginning belong to the previous iteration, and the interleaved dark red tasks and some other blue tasks belong to the iteration we show. At the end, some of the blue tasks overlapped with the pink tasks belong to the iteration we show, while the pink tasks, some other blue tasks and the dark red tasks belong to the next iteration.

The trace of Figure 14 shows very good behavior. The execution was performed using a big problem size. In contrast, Figure 15 was performed using a much smaller problem size. It also shows a single iteration with the overlapping with the previous and following iterations. In this case, to have the same amount of tasks, they must be more fine-grained. In fact, they are too fine-grained, and the task management overhead becomes too much. Specifically, in Figure 15 it is possible to see the white color dominating the trace, meaning that most of the time the cores are idle. Looking at the complete trace, it is possible to see core 0 creating tasks during the entire execution. The other cores execute tasks faster than core 0 can creates them, and that is the reason to see so much white color in the trace.

The scenario with a small problem size is important, considering the strong scaling experiment. Given that we set a fixed problem size and then we increase the number of processors, at each new step the problem size per process will decrease, ending up in scenarios such as the one in Figure 15 or with even smaller problem sizes.

We want to determine how many different granularities we need in this mini-app. Looking at Figure 15 we see a

TABLE II
AVERAGE TASK TIME FOR EACH OF THE TASK TYPES OF HPCCG MINI-APP USING A SMALL PROBLEM SIZE

Task type	Average	Factor	Task type	Average	Factor
HPCSparseMV 🟪	79.25 us	1	**ddot_xx** 🟫	4.45 us	4
ddot_xy 🟥	3.73 us	4	**waxpby_beta** 🟦	6.25 us	4
waxpby_negative_beta 🟨	6.48 us	4			

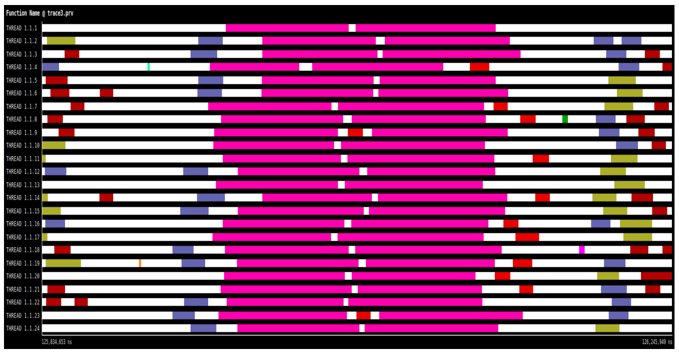
Fig. 16. Execution trace of the LULESH T version using multiple granularities for different task types with a small problem size on 24 cores (1 NUMA socket)

big difference between the pink tasks and the rest. Table II presents the average time a task of a given type requires to complete. Also, each task type has a small square with the same color it has in the trace of Figure 15. Recall that around 100 microseconds is the minimum time we consider a task must last to be worth paying the management costs. None of the task types reaches the given threshold. Accordingly, we should use a higher granularity. However, then, lack of parallelism may appear. Apart from that, *HPCSparseMV* takes much more time to complete than the rest. Consequently, we apply a multiplication factor of 4 to all the task types except *HPCSparseMV* as we show in Table II. We do this to balance the different task times.

After that, there is a ∼1.2x speedup in the execution time. Figure 16 shows what an iteration looks like using the listed factors. Still, there is a lot of white color, meaning cores are running no tasks. This is caused by lack of parallelism: there are not enough tasks to feed all the cores. In consequence, we need to decrease the granularity. However, if we do so, task management overheads may hinder performance. Worksharing tasks offer us the possibility of keeping this granularity, but not be affected by task management overheads, and increasing parallelism due to its internal partitioning of work.

Figure 17 shows the granularity chart using a problem size of 50 elements per dimension. In the x-axis we show the number of tasks/worksharing tasks created per each of the different task types. The y-axis shows the figure of merit (FOM) of the mini-app. The `T` version reaches its peak performance using a reference granularity of 48 tasks per type, and the `TF` version reaches the peak performance using a reference granularity of 1 worksharing task per type. Regarding the implementation, we simply replaced all the regular tasks by worksharing tasks. Figure 17 shows an interesting point. On the right part, when more fine-grained tasks are created, regular tasks outperform

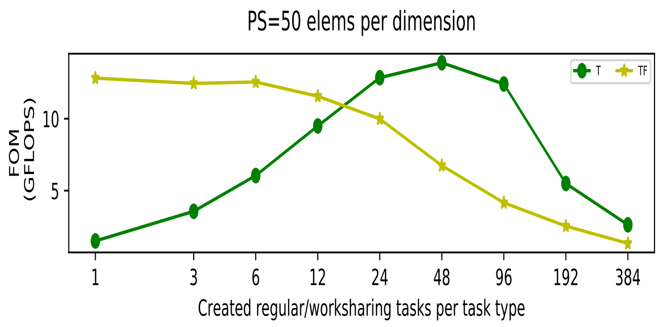

Fig. 17. Granularity chart of HPCCG

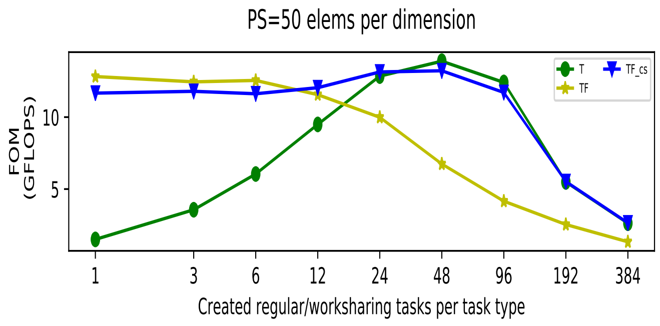

Fig. 18. Granularity chart of HPCCG

worksharing tasks. The problem is that worksharing tasks partition the loop iteration space into as many chunks as cores. In this case, the tasks are already fine-grained, and then they are partitioned even more. As a result, cores end up running really small chunks ($<$ 5 us). Worksharing tasks contain an internal mechanism of synchronization which is very lightweight, but not enough to perform well with such small chunks.

Nevertheless, worksharing tasks offer a mechanism to mitigate this effect. A user can set the minimum chunksize. We have selected a minimum chunksize based on the point where the performance of the `TF` version starts to be worse than the performance of the `T` version in Figure 17. We repeated the granularity analysis, keeping the same problem size (50 elements per dimension), including this new version. The results of the new analysis are shown in Figure 18. After setting the minimum chunksize, both versions behave very similarly in the right-most part, while the `TF` version keeps very good performance in the left-most part.

In this mini-app, unlike LULESH, there is no big difference in the peak performance between the `T` version and the `TF` version using the optimal granularity. In HPCCG, there are not so many tasks and the creator can create tasks rapidly enough for the rest, which was the main issue in the LULESH. Figure 19 shows an iteration of the `TF` version using the optimal granularity. There, it can be seen the main problem that HPCCG presents. In the left part, between the dark red tasks and the blue tasks, there are two very small tasks. This also happens in the right part between the red tasks and the blue tasks. Those small tasks require the data computed by all the previous ones, and the following tasks require the data

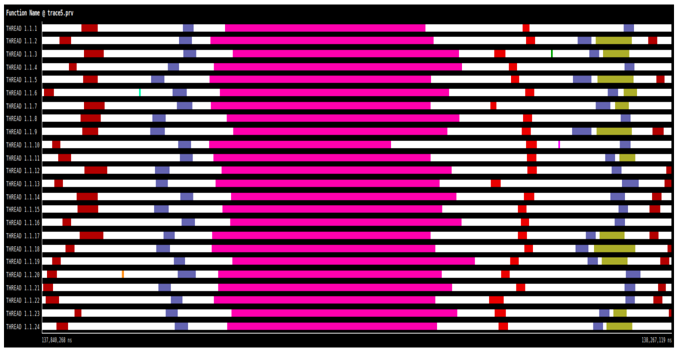
Fig. 19. Execution trace of the HPCCG TF version using a single granularity for all the tasks on 24 cores (1 NUMA socket)

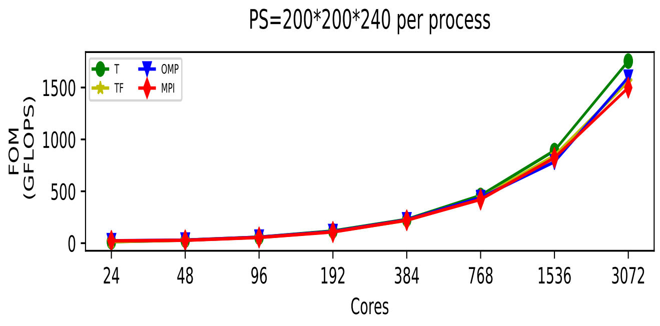

Fig. 20. Results of the HPCCG weak scaling experiment

computed by those two small tasks. Basically, those tasks close the parallelism and open it again, following a fork-join pattern. Apart from that, there are still some other regions with cores running no tasks. The cause is the dependences between different tasks. The runtime requires some time to release the dependences of a task and schedule the new ready tasks. Again, with higher granularities, this effect would be mitigated, but the low problem size prevents us from increasing granularity.

After detailing the modifications done in the implementation, and selecting an adequate granularity, we are able to move forward and perform the experiments in a distributed environment. Figure 20 shows the results of the weak scaling experiment using a problem size of 200*200*240 per process, and Figure 21 shows the results of the strong scaling experiment using a problem size of 200*200*3072.

In the weak scaling scenario shown in Figure 20 all the versions perform very similarly. The T version stands out over the rest, followed by the TF version. The reason for this is the load imbalance introduced by the sparseMV kernel. Different regions of the sparse matrix have different number of non-zeros. Tasks deal better with load imbalance, and are able to get computation and communication overlapping. Overall, it gives the tasking versions a small boost in performance of up to 1.14x and 1.08x for the T and the TF versions, respectively. As a final remark, we were not able to scale to more cores in this scenario because of some overflow problems in the application indexes.

In the strong scaling scenario, all the versions perform very similarly until 3072 cores. Then, the T version cannot scale as much as OMP and TF. Regarding the MPI version, it cannot scale further because the problem cannot be split in more MPI processes. The TF version is competitive until 6144 cores, where it stops scaling, while the OMP version keeps scaling up until the end (12288 cores). With such a number of cores, the problem size per process becomes very small and most of it fits in cache. The OMP version is able to exploit data locality thanks to the static scheduling of the worksharing loops. However, the TF cannot do so because of two reasons: worksharing tasks do not guarantee that the same core executes always the same iterations, and the structures of the runtime pollute the cache.

All in all, the TF version is able to keep very competitive performance until a large number of cores compared to OpenMP, in an application perfectly suited for the fork-join execution. Compared to the T version, this experiment evidences the problems of task granularity that regular tasks suffer, and that worksharing tasks can mitigate, obtaining up to 1.33x speedup.

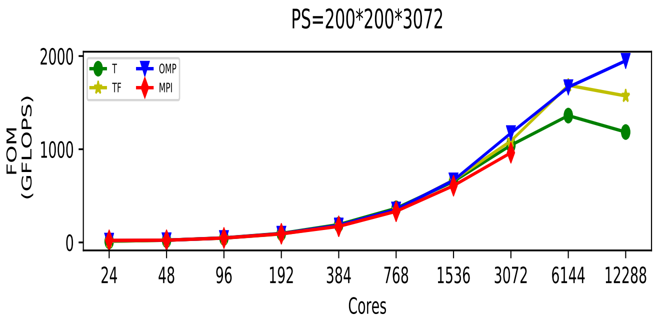

Fig. 21. Results of the HPCCG strong scaling experiment

V. Related work

"MPI+X" has become the dominant paradigm for hybrid parallel programming. This approach is based on the Message Passing Interface (MPI) plus a second approach/model leveraging the system- or the node- level capabilities of the HPC system. To improve interoperability in between these two components (the *MPI* and the *X*), the HPC community is actively exploring new opportunities and extensions. Some of them have been already incorporated into the MPI standard: levels of threading support (introduced in MPI 2.0 [16]), matched-probe operations or inter-process shared memory (both included in MPI 3.0 [17]). Some of these ideas have not yet been included into the standard, but they have proven their usability: InfiniBand GPU-to-GPU communication [18], or the Endpoints [19] extensions. And finally some of them directly impact on the *X* component: the collective offloading at clusters [20], implemented in OmpSs, for instance.

Programming the *X* component may follow, intentionally dismissing the data-parallelism, two different approaches: the fork-join model or the tasking model. The fork-join model efficiently manages the overhead of the computational phase parallelization, but it also imposes very strong restrictions with respect to the thread synchronization (in the join phase). The tasking model (with dependences) allows the data-flow execution by means of taskifying the computation and communication phases and let the dependences to guide the execution. However, it adds a non-negligible overhead that directly impacts in the performance.

- In [5], the authors explore a new type of task directive (directly applied to loops: worksharing tasks), leveraging the benefits of the fork-join model with respect to the low overhead, and removing the synchronization constraints imposed by OpenMP parallel regions. However, the study does not analyze any interoperability option to improve the communication behaviour between MPI processes.
- In [21], [2], the authors turn inside out the interoperability options between the OmpSs-2 programming model and the MPI library. They extend the functionalities of *blocking* and *non-blocking* services allowing the task-based runtime system to context switch when a MPI communication service is not ready yet. This approach minimizes the number of cycles a CPU begins to idle (when there are still other tasks to execute). However, the study is completely based on the pure tasking model, imposing overheads that make it impossible to work with very fine granularities.

In this paper we carried out a study combining both approaches, leveraging the strengths, and minimizing the weaknesses, of each one.

VI. Concluding remarks

Hybrid programming is a promising approach to exploit large clusters. However, the combination of MPI with other shared-memory programming models is not trivial due to correctness (i.e., deadlocks) and performance issues, especially when combining MPI with tasking. Nevertheless, tasks and its data-flow execution model can offer key advantages such as a natural overlap of computation and communication phases, inherent tolerance to load imbalance and fine-grained synchronizations.

In this paper, we have developed a hybrid version of LULESH and HPCCG proxy applications to overcome the correctness and performance issues described in the introduction by combining TAMPI with worksharing tasks. We have fully taskified all computation and communication phases of both applications using OmpSs-2 programming model and TAMPI library. Moreover, after an in-depth analysis of task granularities performed on both applications, we have used OmpSs-2 worksharing tasks to overcome the task granularity issues that naturally arise in some applications, but always show up in strong scaling scenarios done at scale. Our results show significant improvements in LULESH, reaching a speedup of up to 1.3x compared to the best of the other state-of-the-art approaches, and 3x compared to regular tasks. This difference is explained due to the better tolerance of the data-flow execution model to cope with load imbalance, as well as, the lower overhead of worksharing tasks and its ability to feed a large number of cores. Regarding HPCCG, an application with a very regular structure that is perfectly addressed with a fork-join pattern, our implementation reaches 1.08x speedup compared to other state-of-the-art approaches, and 1.33x speedup compared to regular tasks. In light of the results, we conclude that the use of worksharing tasks and TAMPI is a good approach to efficiently exploit large clusters.

VII. Future Work

As future work, we plan to try new applications and benchmarks to check if they can benefit from our approach.

We also plan to introduce a new feature called *taskloop for*. The *taskloop* directive, applied to a *for* loop, creates as many tasks as indicated by the user through the *num_tasks* clause or the *grainsize* clause. Given the similarity between regular tasks and worksharing tasks, we believe that the taskloop directive can be modified to create worksharing tasks instead of regular tasks. Using this directive, users can apply blocking more easily.

Finally, given the importance of properly handling data locality in HPCCG, we plan to add new mechanisms to enable worksharing tasks to exploit data locality better.

Acknowledgment

This project is supported by the European Union's Horizon 2020 Research and Innovation programme under grant agreement No.s 754304 (DEEP-EST) and 823767 (PRACE), the Ministry of Economy of Spain through the Severo Ochoa Center of Excellence Program (SEV-2015-0493), by the Spanish Ministry of Science and Innovation (contract PID2019-107255GB) and by the Generalitat de Catalunya (2017-SGR-1481). The work has been performed under the Project HPC-EUROPA3 (INFRAIA-2016-1-730897), with the support of the EC Research Innovation Action under the H2020 Programme; in particular, the author gratefully acknowledges the support of Dr Mark Bull (EPCC) and the computer resources and technical support provided by EPCC.

References

[1] Barcelona Supercomputing Center, "TAMPI library," accessed: 2020-05-24. [Online]. Available: https://github.com/bsc-pm/tampi

[2] K. Sala, X. Teruel, J. M. Perez, A. Pena, V. Beltran, and J. Labarta, "Integrating blocking and non-blocking mpi primitives with task-based programming models," *Parallel Computing*, 12 2018.

[3] A. Navarro, S. Mateo, J. M. Perez, V. Beltran, and E. Ayguadé, "Adaptive and architecture-independent task granularity for recursive applications," in *International Workshop on OpenMP*. Springer, 2017, pp. 169–182.

[4] Barcelona Supercomputing Center, "OmpSs-2 Programming Model," accessed: 2020-05-24. [Online]. Available: https://pm.bsc.es/ompss-2

[5] M. Maroñas, K. Sala, S. Mateo, E. Ayguadé, and V. Beltran, "Worksharing tasks: An efficient way to exploit irregular and fine-grained loop parallelism," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019, pp. 383–394.

[6] OpenMP Architecture Review Board, "OpenMP Application Programming Interface," November 2018, accessed: 2019-03-24. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[7] Michael A. Heroux, "High Performance Computing Conjugate Gradients: The original Mantevo miniapp," accessed: 2020-05-24. [Online]. Available: https://github.com/Mantevo/HPCCG

[8] Lawrence Livermore National Laboratory, "Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)," accessed: 2020-05-24. [Online]. Available: https://github.com/LLNL/LULESH

[9] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.

[10] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[11] Barcelona Supercomputing Center, "MareNostrum IV User's Guide," Barcelona, Spain, March 2019, accessed: 2020-05-24. [Online]. Available: https://www.bsc.es/support/MareNostrum4-ug.pdf

[12] ——, "Mercurium Compiler," accessed: 2020-05-24. [Online]. Available: https://github.com/bsc-pm/mcxx

[13] ——, "Nanos6 Runtime," accessed: 2020-05-24. [Online]. Available: https://github.com/bsc-pm/nanos6

[14] ——, "Extrae," accessed: 2020-05-24. [Online]. Available: https://tools.bsc.es/extrae

[15] ——, "Paraver," accessed: 2020-05-24. [Online]. Available: https://tools.bsc.es/paraver

[16] Message Passing Interface Forum, "A message-passing interface standard version 2.0," 1997, accessed: 2020-03-24. [Online]. Available: https://www.mpi-forum.org/docs/mpi-2.0/mpi2-report.pdf

[17] ——, "A message-passing interface standard version 3.0," 2012, accessed: 2020-03-24. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[18] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "Mvapich2-gpu: Optimized gpu to gpu communication for infiniband clusters," *Comput. Sci.*, vol. 26, no. 34, p. 257266, Jun. 2011. [Online]. Available: https://doi.org/10.1007/s00450-011-0171-3

[19] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur, "Enabling mpi interoperability through flexible communication endpoints," in *Proceedings of the 20th European MPI Users Group Meeting*, ser. EuroMPI 13. New York, NY, USA: Association for Computing Machinery, 2013, p. 1318. [Online]. Available: https://doi.org/10.1145/2488551.2488553

[20] F. Sainz, J. Bellon, V. Beltran, and J. Labarta, "Collective offload for heterogeneous clusters," in *Proceedings of the 2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, ser. HIPC 15. USA: IEEE Computer Society, 2015, p. 376385. [Online]. Available: https://doi.org/10.1109/HiPC.2015.20

[21] K. Sala, J. Bellón, P. Farré, X. Teruel, J. M. Pérez, A. J. Peña, D. J. Holmes, V. Beltran, and J. Labarta, "Improving the interoperability between MPI and task-based programming models," in *Proceedings of the 25th European MPI Users' Group Meeting, Barcelona, Spain, September 23-26, 2018.* ACM, 2018, pp. 6:1–6:11. [Online]. Available: https://doi.org/10.1145/3236367.3236382