Towards Data-Flow Parallelization for Adaptive Mesh Refinement Applications

Kevin Sala
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
kevin.sala@bsc.es

Alejandro Rico
Arm Research
Austin, TX, USA
alejandro.rico@arm.com

Vicenç Beltran
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
vbeltran@bsc.es

***Abstract*—Adaptive Mesh Refinement (AMR) is a prevalent method used by distributed-memory simulation applications to adapt the accuracy of their solutions depending on the turbulent conditions in each of their domain regions. These applications are usually dynamic since their domain areas are refined or coarsened in various refinement stages during their execution. Thus, they periodically redistribute their workloads among processes to avoid load imbalance. Although the defacto standard for scientific computing in distributed environments is MPI, in recent years, pure MPI applications are being ported to hybrid ones, attempting to cope with modern multi-core systems. Recently, the Task-Aware MPI library was proposed to efficiently integrate MPI communications and tasking models, providing also the transparent management of communications issued by tasks.**

In this paper, we demonstrate the benefits of porting AMR applications to data-flow programming models leveraging that novel hybrid approach. We exploit most of the application parallelism by taskifying all stages, allowing their natural overlap. We employ these techniques on the miniAMR proxy application, which mimics the refinement, load balancing, communication, and computation patterns of general AMR applications. We evaluate how this approach reduces the time in its computation and communication phases while achieving better programmability than other conventional hybrid techniques.

***Index Terms*—Adaptive Mesh Refinement, AMR, MPI, Tasks, Data-Flow, OpenMP, OmpSs-2, TAMPI, miniAMR**

I. Introduction

As the exascale era approaches, the HPC systems are further evolving to enormous machines offering tens of thousands of high-end computing nodes, each of these featuring a large number of processing units. Those systems connect their nodes using high-bandwidth and low-latency interconnection networks, allowing high scalability efficiency across thousands of nodes. Nevertheless, achieving the full utilization of their resources has become a severe challenge for the HPC community. Most scientific applications are parallelized using only MPI [1], and although it is the defacto programming model in distributed environments, it is not the best option to exploit the intra-node parallelism in modern multi-core and many-core processors. Pure MPI approaches can make applications less flexible while becoming more sensitive to load imbalance or differences among processing unit's throughput.

ORCID: 0000-0001-8233-1185, 0000-0003-1282-8887, 0000-0002-3580-9630

The most widespread solution is the hybrid parallel programming technique [2] [3] [4], where distributed and shared-memory programming models are combined to exploit both inter- and intra-node parallelism. Hybrid applications usually combine MPI with OpenMP [5], which is the leading API for shared-memory parallelism. Hybrid programming is already a conventional technique but is considered a crucial component to scale in future exascale systems [6]. Commonly, MPI+OpenMP applications parallelize their computation phases while keeping their MPI communications serialized, in what is called a fork-join approach. In general, this simple technique does not scale nor permit the overlapping of phases. There are more advanced techniques, like taskifying both computation and communication phases and connecting tasks through data dependencies. This allows removing global synchronizations by means of inter-task dependencies across processes, which is crucial for the performance of parallel applications at large scale [7]. However, MPI and OpenMP do not currently provide support to implement this approach safely and efficiently. They were not designed to be combined and only provide mechanisms to coexist on an application.

The Task-Aware MPI (TAMPI) library [8] [9] was recently proposed to overcome these limitations. This library integrates blocking and non-blocking MPI operations with tasking programming models, like OpenMP and OmpSs-2 [10], allowing the efficient execution of MPI operations from within tasks. TAMPI is responsible for managing the progress of MPI communications issued by tasks, whereas the tasking model inherently provides the overlap between tasks of different phases. This way, application developers can focus on exposing their application's parallelism without being concerned about low-level aspects.

Recently, some applications have been studied to leverage this hybrid data-flow model, as the Gauss-Seidel and IFSKernel [8] benchmarks, and the CREAMS [11] fluid dynamics application. However, there are no studies on applying this approach to more dynamic algorithms featuring periodic load balancing or modifications on the domain space, forcing the application to increase/decrease the problem complexity during its execution. Applications leveraging the Adaptive Mesh Refinement method [12] match those conditions.

The AMR is a popular method used by several distributed-memory simulation applications that adapt the accuracy of

their solutions depending on the turbulence conditions in each of their domain's regions. Usually, turbulence conditions are produced by the presence of objects from the simulation in some regions of the domain. These applications increase their accuracy in these turbulent regions while keeping the rest with lower accuracy. This method saves both computational and memory resources compared to static grid approaches where the whole domain is uniformly refined to the same resolution. These applications present several opportunities for improvement. Their computation and communication patterns dynamically change depending on the simulated problem due to the periodic refinement stages, which may increase or decrease the accuracy in some particular areas. Also, refinement stages alter the number of areas that each process is responsible for, meaning that their workload may vary unequally. Thus, they must perform load balancing stages regularly to avoid imbalance among processes.

MiniAMR [13] [14] is a proxy application that mimics the communication, refinement, and load balancing stages of larger applications, like CTH [15]. This mini-app aims at facilitating the task of testing new programming techniques and new computer architectures. Recently, Rico et al. [16] proposed and implemented a taskification approach for miniAMR. They ported the mini-app to the OpenMP tasking model to be evaluated on a single node. Since they focused on the shared-memory parallelization, they disabled the MPI communication phases. That makes this application an excellent opportunity to apply the data-flow model that TAMPI offers. We aim at having a complete taskification of miniAMR such that the tasks, connected through data dependencies, conduct all computation and communication phases, and leveraging TAMPI to transparently manage MPI operations issued by tasks. This way, tasks from different phases will be able to overlap naturally, without sacrificing programmability.

In this work, (1) we propose a complete hybrid data-flow taskification for AMR applications comprising their computation, communication, refinement, and load balancing phases, with the objective of enhancing their programmability and performance; (2) we explain the porting of miniAMR to this approach step by step, and (3) we provide an in-depth performance and scalability comparison between our approach, the MPI-only reference implementation, and a MPI+OpenMP fork-join variant, using up to 12288 cores.

II. Background

A. Adaptive Mesh Refinement, MiniAMR

Scientific and engineering applications that incorporate Adaptive Mesh Refinement, as CTH [15], are usually large applications featuring numerous options and modes. They are too complex for evaluating new programming techniques or new system architectures. MiniAMR [13] [14] is a mini-app included in the Mantevo suite [17] that addresses those limitations. This proxy application mimics the refinement, coarsening, load balancing, communication, and computation patterns found in general AMR applications. The reference implementation [14] of miniAMR is based only in MPI, but they also provide an experimental fork-join hybrid variant that combines MPI and OpenMP threads [14].

```
1  foreach timestep or simulation time finished do
2      foreach stage per timestep do
3          foreach communication var group do
4              communicate(group);
5              stencil(group);
6              if checksum stage then checksum(group);
7          end
8      end
9      if refinement timestep then refine();
10 end
```

Algorithm 1: The main loop of miniAMR.

The physical domain of miniAMR is a rectangular mesh representing a unit 3D cube. The mesh is divided into blocks along all three dimensions. The application provides options to configure the number of MPI processes, the initial number of blocks per process, and the block size, all of them in each dimension. Therefore, each MPI process holds a specific initial number of blocks, and that initial configuration represents the coarsest level of the mesh.

MiniAMR provides options to define multiple input objects, which are simulated during the application's execution. There are up to 16 types of objects: rectangles, spheroids, cylinders, etc. Each object specifies its initial position, initial size, movement rate, growth rate, and whether the object should bounce when colliding with the mesh boundaries. MiniAMR considers all these input objects when deciding which regions of the mesh need more precision and should be refined. For instance, blocks containing boundaries of an object could be refined into finer blocks. The refinement of a block consists of doubling its resolution in each dimension by splitting that block into eight new blocks. Although each block has a specific refinement level, all have the same size (or number of grid cells), causing the most refined blocks to provide a higher simulation accuracy. Conversely, the coarsening of blocks produces a consolidation of eight refined blocks into a single coarser, reducing the simulation accuracy in that region. Then, communications, either inter- or intra-process, involve exchanging the block boundary faces (or ghost values) between neighboring blocks. To reduce the complexity of these communications, miniAMR forces neighboring blocks to be at a maximum difference of one refinement level.

Grid cells are comprised of a user-defined number of variables, and stencil algorithms are independently applied to the variables in each cell. There are several stencils, but we focus on the 7-point stencil, which computes the value of a cell as the average of itself and its six 3D neighboring cells.

Algorithm 1 shows the main loop of the miniAMR executed by each MPI process. The mini-app runs during a specific number of timesteps or during a maximum simulation time. Timesteps are divided into various stages, where it takes place the communication of neighboring block faces, and then, the computation applying the stencil algorithm to all process' blocks. Communication and stencil functions handle cell variables by communication variable groups, where users specify the number of groups to use. Each `communicate` and

```
1  foreach dir in X, Y, Z directions do
2      foreach proc in neighbors[dir] do
3          foreach face in recvfaces[dir][proc] do
4              MPI_Irecv(recvbuf[face]);
5          end
6      end
7      foreach proc in neighbors[dir] do
8          foreach face in sendfaces[dir][proc] do
9              pack_face(face, sendbuf[face]);
10             MPI_Isend(sendbuf[face]);
11         end
12     end
13     intraprocess_communication();
14     while any recvbuf[*] pending do
15         id = MPI_Waitany(recvbuf[*]);
16         face = recvfaces[id];
17         unpack_face(face, recvbuf[face]);
18     end
19     MPI_Waitall(sendbuf[*]);
20 end
```

Algorithm 2: The `communicate` function.

`stencil` call handles a single group of variables, however, in realistic simulations, there is usually a single group.

The `communicate` function performs both the intra- and inter-process exchanges of boundary block faces (ghost values) among neighboring blocks. Algorithm 2 shows the pseudocode of this function. Exchanges are performed in a single direction at a time. The loop at line 2 starts receiving the required boundary faces for every remote neighbor process in that direction. Then, the loop at line 7 packs each of the boundary faces that must be sent to a neighbor. The packing consists of copying the face data from the mesh block to a contiguous communication buffer. Once packed, at line 10, the face data stored in the sending communication buffer is sent through MPI. While the MPI communications are in-flight, at line 13, the intra-process exchanges take place. That exchange involves the local copy of the bordering faces for each pair of neighboring blocks that are both stored in the current MPI process. Then, the loop at line 14 iterates as long as there are pending receivings. Once a face is received, at line 17, the mini-app unpacks the face data that is stored in the receiving communication buffer. The unpacking consists of copying the face data from the receiving buffer to the corresponding mesh block. Finally, at line 19, it waits until all sending operations complete before starting the exchanges for the next direction.

The code shows that each face is sent/received in a separate message, although that behavior is only enabled by the `--send_faces` option. By default, all faces exchanged with a remote process are grouped into a single MPI message. Communications are always done using non-blocking MPI operations, trying to overlap intra- and inter-process communications. Send/receive operations for the same direction can be performed concurrently because each face has a specific section in the global send/receive buffer. However, directions must be processed sequentially since they use the same communication buffers. MPI messages are tagged with the face's identifier, which both sender and receiver know beforehand.

Back in the main loop, miniAMR checks the correctness of the solution every specific number of stages. The checksum involves a local array reduction over all variables, followed by a global array reduction over the arrays of all processes. Furthermore, every specific number of timesteps, the mini-app moves the objects based on their movement rate, and then, it carries out a refinement stage. During that stage, blocks of the mesh may be refined or coarsened, considering the characteristics of the intersecting objects.

The miniAMR that we have explained is the MPI-only version that we will use in the evaluation to compare with our hybrid approach. This MPI-only baseline corresponds to the reference implementation [14] with two changes introduced by Rico et al. [16]. The first change is that the `stencil` function handles a group of variables instead of a single variable. Secondly, each block has a pointer to a contiguous array storing all its variables for the three dimensions, instead of having disaggregated arrays for each variable and dimension.

B. Hybrid Parallel Programming

The MPI [1] and OpenMP [5] standards are the programming models par excellence for scientific and engineering applications in distributed environments. MPI is the principal message-passing standard that targets the inter-node parallelism. OpenMP is the main portable API for shared-memory parallelism. They are usually combined in applications to get the benefits from both models, in what is called hybrid parallel programming. Although it is not a novel technique, it is still one of the key elements for efficiently leveraging the resources of current and future exascale systems [6].

OpenMP offers different ways to define parallelism, but the main two are through the thread fork-join and tasking models. The fork-join model allows defining parallel regions and loops that are concurrently executed by multiple threads, which are managed by an underlying runtime library. In contrast, the tasking model allows to define tasks, which are independent pieces of code that are scheduled and executed asynchronously. Users define the input and output data for each task and the runtime library tries to run them in parallel respecting their execution order constraints. As an alternative, OmpSs-2 [10] is the second generation of the OmpSs [18] programming model and defines a full tasking model with advanced dependency features.

The most common practice in hybrid applications is the combination of MPI and OpenMP in a fork-join approach. Computation phases are parallelized with OpenMP, while MPI communication phases are performed sequentially. The parallelism is closed before starting a communication phase, or just a small portion is parallel. Although it is a simple approach, it generally does not scale as well as the MPI-only version. A more advanced approach is the taskification of both computation and communication phases. Task data dependencies connect both task types and guarantee a correct execution order. In this way, the taskification offers a natural overlap of computation and communication phases, which is a challenging objective in other approaches. This strategy follows the data-flow paradigm principles, where computational processes are broken into individual work units handling smaller portions of data concurrently. However, MPI

and OpenMP were not designed to be combined efficiently. In fact, MPI only defines the MPI threading levels [1] to interact with other parallel programming models.

The Task-Aware MPI library [8] [9] was recently proposed to overcome these limitations. TAMPI allows the safe and efficient taskification of MPI communications, including all blocking, non-blocking, point-to-point, and collective operations. The TAMPI library allows calling MPI communication operations from within tasks. Blocking MPI operations pause the calling task until the operation completes, but in the meantime, the tasking runtime system may execute other tasks.

Non-blocking operations can be bound to a task through two API functions named `TAMPI_Iwait` and `TAMPI_Iwaitall`, which have the same parameters as the standard `MPI_Wait` and `MPI_Waitall`, respectively. These two TAMPI functions are non-blocking and asynchronous and can be used to bind the completion of the calling task to the finalization of the MPI requests passed as parameters. The calling task will not release its dependencies until (1) the task finishes its execution and (2) all bound MPI operations complete. Its successor tasks will be ready to execute only once its dependencies are released.

We must highlight that a task can call those TAMPI functions several times, binding multiple MPI requests during its execution. Moreover, TAMPI offers a wrapper function for each non-blocking MPI operation, which performs the standard non-blocking operation, and then, it automatically calls `TAMPI_Iwait` with the resulting MPI request. The `TAMPI_Isend` and `TAMPI_Irecv` are an example.

III. Related Work

Since the Adaptive Mesh Refinement method [12] was proposed, several studies have been conducted on AMR applications and their parallelization. Different hybrid fork-join MPI+OpenMP approaches were studied [19] [20]. Recently, a MPI+OpenMP stealing mechanism was proposed by Samfass et al. [21] that enables distributed work-stealing among processes to avoid the common load imbalance in AMR applications. A communication thread per rank is used to coordinate the work-stealing, while OpenMP tasks conduct computations. Prat et al. [22] studied the taskification of computations in AMR applications using OpenMP tasks and dependencies, combined with cache blocking and vectorization techniques. However, they did not include the study of communication patterns. As previously mentioned, miniAMR [13] was proposed to facilitate the testing of new programming techniques and architectures. Rico et al. [16] ported miniAMR to OpenMP tasks, focusing only on single node executions. In this paper, we take their parallelization as the base for our full taskification of miniAMR.

Several frameworks provide APIs for building parallel AMR applications. AMReX [23] is a framework based on the ideas of BoxLib [24] and Chombo AMR [25], which facilitates the building of block-structured AMR applications and supports hybrid MPI+OpenMP parallelism. AMReX divides the domain into smaller boxes and distributes them among MPI ranks transparently. Each rank is responsible for computing its boxes. These computations can be parallelized at two levels: a box is entirely computed by a single OpenMP thread, or it can be processed by multiple threads. Zhang et al. [26] presented the idea of tiling in AMR, where they logically divide each box into tiles of a specific size. Each tile is then computed by a thread, so multiple tiles can be processed in parallel.

Farooqi et al. [27] extended the AMReX framework with an API to support the asynchronous execution of AMR computation phases, which can overlap the internal AMReX communications. They implement a runtime system that manages the dependencies among AMR phases and their required communications. They use a few threads dedicated to performing communications, along with other threads running the computation phases. This API could probably be implemented using OpenMP/OmpSs-2 and TAMPI, leveraging task dependencies to overlap computation and communication tasks transparently. This way, AMReX would not need to implement the management of parallelism and dependencies. Also, the dedicated communication threads would disappear because communication tasks would run in parallel with other tasks in any available core.

Finally, some applications were already studied to benefit from the TAMPI [8] features. The Gauss-Seidel and IFSKernel [8] benchmarks were ported to a similar approach that we are studying in this paper for AMR. Also, the CREAMS [11] computational fluid dynamics application was studied and ported to TAMPI+OmpSs-2. These three applications achieved better scalability than the rest of their non-TAMPI variants.

IV. Data-Flow Parallelization for MiniAMR

In this section, we describe the data-flow parallelization that we propose for miniAMR. We focus on exploiting most of the mini-app parallelism by taskifying the heaviest parts: the intra- and inter-process communication, stencil, checksum, and some refinement and load balancing sections. Our objective is to design an approach that combines tasks across different stages through data dependencies, avoiding implicit and explicit barriers, which could break the data-flow execution model and limit the potential parallelism. That allows us to overlap computation and communication tasks from the same phase but also between different phases. Although we explain the parallelization for miniAMR, most transformations should apply to other larger AMR applications.

We take as a base the taskification of the intra-process code sections made by Rico et al. [16]. They taskified the stencil, checksum, and intra-process communications with OpenMP tasks. They disabled the MPI communication part, as they evaluated miniAMR only on a single node. Thus, they modified the `stencil`, `checksum`, and `communicate` functions from Algorithm 1 to instantiate tasks to process the blocks of the mesh. Firstly, the `stencil` function creates a task per block, declaring an input-output dependency on the block's data section that corresponds to the group of variables being processed. In this case, blocks are computed in parallel. Secondly, they applied a similar approach in `checksum`, where the local reduction is performed in parallel

```
1  foreach dir in X, Y, Z directions do
2      foreach proc in neighbors[dir] do
3          foreach face in recvfaces[dir][proc] do
4              #pragma omp task out(recvbuf[face])
5              TAMPI_Irecv(recvbuf[face]);
6          end
7      end
8      foreach proc in neighbors[dir] do
9          foreach face in sendfaces[dir][proc] do
10             #pragma omp task in(face) out(sendbuf[face])
11             pack_face(face, sendbuf[face]);
12             #pragma omp task in(sendbuf[face])
13             TAMPI_Isend(sendbuf[face]);
14         end
15     end
16     intraprocess_communication();
17     foreach proc in neighbors[dir] do
18         foreach face in recvfaces[dir][proc] do
19             #pragma omp task out(face) in(recvbuf[face])
20             unpack_face(face, recvbuf[face]);
21         end
22     end
23 end
```

Algorithm 3: The taskified `communicate` function.

using a task per block. These tasks only read the block's data, thus they directly depend on the updates made by `stencil` tasks. Next, parallelism is closed with an OpenMP `taskwait` after the local reduction, and then, the global MPI reduction takes place across all processes. Then, they modified the `communicate` function, previously shown in Algorithm 2, by disabling the pack/unpack of boundary block faces and their remote transmission. They taskified the intra-process communication, which corresponds to the local copies between neighboring blocks in the same process. As all these tasks are connected through data dependencies, they achieved the overlapping of these three phases. Parallelism was only closed during the stages with a checksum and before starting a refinement phase. Taking their taskification as a base, in the following subsections, we describe our parallelization strategy step by step for all the mini-app phases.

A. Inter-Process Communication

Our first step is to integrate the inter-process communication into the current taskification, such that MPI communication tasks naturally combine along with the rest of existing tasks. In this step, we will leverage the features offered by the TAMPI library [8] [9], explained in Section II-B. Since the reference version of miniAMR already leverages non-blocking MPI communication, we use the TAMPI support for non-blocking operations, e.g., `TAMPI_Isend`.

Algorithm 3 shows the taskification of the miniAMR's communication phase. The structure of the function is similar to the one from the MPI-only baseline. We start by taskifying the receiving of faces at line 4. Since the receive operation stores data into the receiving buffer, we declare an output dependency on that buffer. We substitute the `MPI_Irecv` for a `TAMPI_Irecv` call, such that the task performs a standard `MPI_Irecv` and binds the task completion to the finalization of the resulting MPI request. As explained in Section II-B, the TAMPI function is non-blocking and asynchronous, so the task may finish its execution before receiving the data. Therefore, the data must not be consumed inside the task. Once the receive operation finishes, the task will complete and release its data dependencies, marking its successor tasks as ready to execute. Only at this point, the content of the buffer can be safely consumed.

Our taskification continues at the sending loop, at line 9, where we instantiate a task for each packing and sending function call respectively. Firstly, the `pack_face` function copies the face data from the mesh block to the corresponding sending buffer. Hence, the packing task must declare an input dependency on the face, located in a mesh block, and an output dependency on the sending buffer section. Then, the buffer's data is sent by a task through `TAMPI_Isend`. In this case, the sending task declares an input dependency on the buffer because MPI reads it. This task will fully complete when the send operation finishes, meaning that it is safe to reuse the buffer. The next phase, at line 16, is the intra-process communication, where tasks perform copies among local neighboring blocks. This part was already taskified by Rico et al. [16], thus we have not modified it. We made sure that tasks from both intra- and inter-process communication (local copy and TAMPI tasks) are safely combined and connected through data-flow dependencies.

The last phase is the unpacking of the received faces, at lines 19-20. Once a face is received, and the corresponding receiving task (line 4) completes, another task can unpack the data. The `unpack_face` function copies the face data from the receiving buffer to the corresponding mesh block. Consequently, an unpacking task must declare an input dependency on the receiving buffer section and an output dependency on the face, which is located in a mesh block.

Notice that we removed all calls to `MPI_Waitany` and the complexity of managing asynchronous MPI requests disappeared. In our approach, TAMPI transparently manages the MPI operations and their progress, while the tasking model naturally grants the overlapping of phases. Hence, we can completely focus on exposing the application's parallelism instead of being concerned about low-level aspects.

These tasks are instantiated for each of the X, Y, and Z directions, but miniAMR uses the same communication buffer space for all directions. That can reduce the parallelism when sending/receiving faces through MPI because tasks from different directions may depend on the same buffer section. To address this issue, we introduce a new option called `--separate_buffers`, which reserves a send/receive buffer for each direction. This way, tasks from different directions use distinct communication buffers, preventing any false dependency. This option is convenient for tasking models that do not support region dependencies, since buffer section sizes may differ among directions. Tasking models featuring region dependencies (e.g., OmpSs-2) would support that case naturally, but with OpenMP, we would need to express the tasks' dependencies using multi-dependencies [5].

The structure shown in Algorithm 3 represents the behavior when `--send_faces` is enabled. However, as mentioned in Section II-A, the default is to communicate all faces of a

```
1  foreach timestep or simulation time finished do
2      foreach stage per timestep do
3          foreach communication var group do
4              communicate(group);
5              stencil(group);
6              if checksum stage then checksum_local(group);
7          end
8          if checksum stage then
9              #pragma omp taskwait
10             checksum_remote(all groups);
11         end
12     end
13     if refinement timestep then
14         #pragma omp taskwait
15         refine();
16     end
17 end
```

Algorithm 4: The main loop in our taskified miniAMR.

direction and neighbor arranged into a single MPI message. We support that case by instantiating a single communication task per direction and neighbor. Now, a sending task may depend on multiple packing tasks because each of these latter writes to a section of the sending task's buffer. Thus, we express its dependencies on all its buffer sections with OpenMP/OmpSs-2 multi-dependencies. The same occurs with the receiving tasks and their succeeding unpackers.

Regarding the number of communication tasks, we have added another option called `--max_comm_tasks` that expects a non-negative integer. It is only considered when enabling `--send_faces` and indicates the maximum number of communication tasks (thus messages) per direction and neighbor. This option allows us to control the parallelism exposed in this phase. The default value is zero and means that we create a communication task and message per face. Generally, four or eight communication tasks (messages) per direction and neighbor are sufficient to obtain satisfactory parallelism. Since computation tasks usually have finer granularities than communication ones, when a communication task completes, it satisfies the data dependencies of multiple computation tasks. Once again, we use multi-dependencies to implement this feature.

Lastly, communication tasks must use a well-defined space of MPI tags to identify face messages. As mentioned in Section II-A, the MPI-only baseline uses face identifiers as MPI tags, known by both the sender and receiver beforehand. In our case, we can continue using face identifiers to compute MPI tags. However, since communication tasks from different directions may run in parallel, we must define a distinct tag space for each of the three directions. For simplicity, we divide the tag space into three sub-spaces, allowing a sufficiently large arbitrary number of tags per direction. Notice we could use other mechanisms instead, like using a different MPI communicator per direction.

B. Refinement & Load Balancing

The next crucial phase is refinement, where mesh blocks are refined or coarsened depending on the objects moving across the mesh. This phase may also perform a load balance stage to equilibrate the number of mesh blocks per process. The refinement phase is performed every specific number of timesteps, and before starting this phase, we have an explicit barrier (`taskwait`) that waits until all previous tasks finish. The call to the refinement function is shown at the end of Algorithm 4. In our experiments, the time spent in this phase does not take more than 10% of the total execution time, however, the percentage significantly grows when the rest of the application is parallelized. Notice that the MPI-only version implicitly parallelizes the refinement because it executes with more ranks (e.g., one rank per core) than hybrid runs (e.g., one rank per socket). There are fewer mesh blocks per rank in MPI-only because the refinement work is naturally divided across all ranks.

The taskification for this stage must be designed smartly to compete with the inherently parallel MPI-only version. This phase is not easy to parallelize because most of the code operates with control data structures. Nevertheless, some code sections work with mesh block data and are reasonably parallelizable. We first focused on profiling this phase, and then, we taskified the most time-consuming parts. One of these heaviest sections is when a mesh block is refined and split into new eight finer blocks. Similarly, another time-consuming section is the consolidation of eight refined blocks into a coarser block. The time spent in these two sections is around 25% of the total refinement time. In the splitting section, the original block data is copied to finer blocks, while in the coarsening section, the eight blocks are copied back to the coarser. We taskify these copies since they can run in parallel.

There is another function that takes up to 70% of the refinement time. The `exchange` function transmits mesh blocks that were marked to be moved to other processes using point-to-point MPI operations. The function is complex because its algorithm needs both control and data MPI messages to coordinate the exchange operation. The source and destination of each block are known beforehand. The receiver must send an acknowledgment (or ACK) message to the sender, indicating whether it has space for the block. Once the sender receives a positive ACK message, it packs and sends the corresponding mesh block using `MPI_Send`. In turn, the receiver waits for the block and unpacks it once it arrives. The `exchange` function may return with blocks pending to exchange when the available space in the MPI processes is very limited, so a subsequent call to the function is required.

The packing, unpacking, sending, and receiving of blocks can be taskified with dependencies, and we can communicate blocks through non-blocking TAMPI, as in Section IV-A. The main thread sends/receives all control messages sequentially, while tasks handle the heavy work. We close the parallelism before returning from the `exchange` function, once we have exchanged the required blocks. In our taskification, we add an extra control message to facilitate the offloading of mesh block data communications. After receiving a positive ACK from the receiver, the sender sends the block identifier as a control message. This way, both processes know the block identifier before starting the data exchange. We use that identifier to tag the MPI message containing the data. We

continue using standard blocking MPI operations for control messages (sequentially issued by the main thread) to reduce their latency, as in the reference MPI-only version.

Our taskification strategy removes nearly 80% of the total refinement time compared to our previous sequential refinement. Although this stage is not fully parallelized, we obtain reasonable performance compared to MPI-only.

C. Checksum & Validation

Our last step is to reduce the impact of the remaining explicit barriers (`taskwait`). One of the barriers is located before a checksum validation, which are performed every specific number of stages. The checksum process, shown in Algorithm 4, is divided into two functions. Firstly, we call `checksum_local` for each group of variables, where several tasks are created to perform the local checksum reduction. Then, before ending that stage, we wait until all instantiated tasks from `communicate`, `stencil`, and `checksum_local` complete (line 9). Lastly, we call the `checksum_remote` function that performs the MPI reduction across all processes and validates the reduced values. After the validation, the loop of stages keeps iterating and creating tasks from the other phases until the next stage with checksum. Our checksum strategy differs slightly from the taskification of Rico et al. [16]. In our case, we close the parallelism after finishing the whole stage, instead of closing it after processing each group of variables. This way, we reduce the number of taskwaits when there are multiple groups.

We can still reduce even more the impact of the explicit barriers thanks to an OmpSs-2 feature: the `taskwait` with dependencies [10]. The idea is to validate the last checksum stage instead of the current one, delaying the true data dependency between the stencil and the barrier at line 9 in Algorithm 4. For this purpose, we leverage two checksum data structures: one for the current checksum stage and another for the previous one. In a stage with checksum, the tasks from `checksum_local` are instantiated to perform the local reduction on the current checksum structure. However, the `taskwait` blocks until the structure of the previous checksum stage can be consumed and validated. Therefore, we avoid closing all the parallelism by waiting for the previous checksum and allowing the next stages to start executing. Notice that if the validation process detects an error, we will abort the program after executing some more stages.

D. Task Granularities

At this point, we have already parallelized the most time-consuming phases following a data-flow strategy. Our tasks ensure that concurrent work from different phases will naturally overlap during the execution. The last point to discuss is task granularities. Most tasks work at the level of a mesh block and for a specific range of variables in that block, as the `stencil`, `communicate`, and `checksum_local` tasks. Thus, their granularity is directly affected by the block size and the number of variables per group specified with `--comm_vars`. We will show in the next section that we did not use more than one group because we already had a reasonable granularity. Additionally, we feature other options to tune the granularity of communication tasks. These are the new `--separate_buffers` and `--max_comm_tasks`, but also the original `--send_faces`, which are explained in Sections IV-A and II-A, respectively. These options determine the granularity of communication tasks, and thus, the parallelism exposed during communication stages.

We must highlight that our tasks declare their dependencies on the range of variables in the block that they are processing, independently of which block parts are accessing. For simplicity, we do not distinguish between depending on a single face or the whole block. Dependencies only consider the mesh blocks and their range of variables, not faces nor geometric subsets. We also want to remark that our strategy focuses on taskifying high-level functions but, if there was not enough parallelism, we could easily use task nesting to uncover more.

V. Evaluation

In this section, we evaluate how our data-flow approach behaves when compared to the reference MPI-only and a conventional MPI+OpenMP fork-join approach. We first describe the evaluation environment, and then, we explain the performance results we have observed for each version. We perform the evaluation on the Marenostrum4 supercomputer, which is located at the Barcelona Supercomputing Center. Each node has two sockets Intel Xeon Platinum 8160 with 24 cores each, working at 2.10GHz, featuring 48 total cores, and 96 GB of main memory per node. We scale up to 256 nodes in our scaling experiments, which corresponds to a maximum of 12288 total cores. Throughout all our evaluation, we use GCC 7.2.0 and Intel MPI 2017.7 for all variants. All programs are compiled with the maximum optimization options available for this architecture, including floating-point optimizations. We use the OpenMP implementation provided by GCC 7.2.0, OmpSs-2 2020.06, and TAMPI 1.0.1.

We will evaluate three different variants of miniAMR. Firstly, **MPI-only** is the pure MPI version that we explained in Section II-A. It is composed of the MPI-only reference [14] after applying the data and code structure modifications proposed by Rico et al. [16]. This version always leverages one rank per core; 48 ranks per node in Marenostrum4. Secondly, **MPI+OMP** fork-join is the experimental hybrid variant included in the official miniAMR repository [14] that combines MPI and OpenMP threads. This version parallelizes the `stencil` computations, the intra-process communication and block face packing/unpacking in `communicate`, and the local checksum reduction. To carry out a fair comparison against our taskified approach, we have parallelized also the splitting/coarsening of blocks and the block packing/unpacking in the refinement phase. All its parallel regions use OpenMP `for` directives with static scheduling. Notice that since this variant follows a fork-join strategy, all its MPI communications are executed by the master thread. Lastly, **TAMPI+OSS** is the hybrid approach that we propose in this paper, which combines MPI, OmpSs-2, and TAMPI. By leveraging OmpSs-2, we can

benefit from the checksum optimization that we proposed in Section IV-C.

Another relevant aspect to discuss is the input objects for the miniAMR simulations. Throughout this section, we leverage two input problems, both of them taken from previous miniAMR articles. The first was used in the work by Rico et al. [16] and represents a big sphere that starts from outside the mesh. During the execution, the sphere enters the mesh from a lower corner provoking the refinement of the intersecting mesh regions. The second input problem is taken from the work by Vaughan et al. [13] and represents four spheres moving inside the mesh. Two spheres are positioned in one side of the mesh and move along the positive X-axis, while the other two spheres are in the opposite side and move in the negative direction. Their position and size allows for them to get close with each other when approaching the center of the mesh without colliding. We compute their movement rate based on the timesteps that we want to run, such that they arrive at the opposite side of the mesh without reaching its borders. Note that the first input may produce a bit more imbalance across processes during the first simulation timesteps because the sphere enters the mesh from a corner, provoking the processes responsible for that area to refine their blocks.

A. Configuration of Ranks per Node

The first step is to find the optimal configuration of ranks per node for the hybrid variants. As previously mentioned, we will execute the MPI-only variant always with 48 ranks per node, filling each compute node. Hybrid variants can execute with different configurations, leveraging more or fewer ranks per node. When changing the number of ranks in each node, we are also modifying the number of cores per rank that OpenMP/OmpSs-2 can leverage. To find the most suitable configuration, we run the single sphere input on four nodes, which is enough to extract conclusions in this step. We simulate that problem during 20 timesteps, with 60 stages per timestep, blocks of 18x18x18 grid cells, and each cell holding 60 variables. We perform a refinement stage every five timesteps and a checksum validation every ten stages. In the case of TAMPI+OSS, we also enable the `--send_faces` and `--separate_buffers` options to expose all the parallelism in the communication phases. Table I shows the execution times of both hybrid variants when varying the ranks per node. We evaluate from 1 rank/node (48 cores/rank) to 16 ranks/node (3 cores/rank). We show for each variant the total execution time, the time spent in the refinement phase, and the time in the rest of phases, labeled as No Refine.

In all our experiments, we have tried to place consecutive ranks and OpenMP/OmpSs-2 threads of the same rank in adjacent cores at the same NUMA domain. As expected, the worst configuration for both variants is one rank/node because Marenostrum4's nodes have two NUMA domains, one per socket. Having a rank executing in two sockets usually performs worse due to NUMA effects. For MPI+OMP, the first configuration that gets a reasonable performance on both total time and non-refinement time is 4 ranks/node. The next two configurations get slightly more performance mainly due to the refinement time, which may vary between different configurations. The 8 ranks/node configuration takes a bit more time in the non-refinement part than the 4 ranks/node, while the last one is too extreme since it only uses three cores per rank. For these reasons, we have decided to use 4 ranks/node from now on, which is a reasonable configuration when there are two NUMA domains. For TAMPI+OSS, we have decided to run with 2 and 4 ranks/node since both perform similarly. Although 2 ranks/node is a valid configuration, we have seen in other experiments that it can be worse during the refinement phase. This is because the refinement is not fully parallelized with OmpSs-2/OpenMP and the refinement work per rank is naturally reduced as the number of ranks increases.

TABLE I
THE TIME (S) VARYING THE NUMBER OF RANKS PER NODE ON 4 NODES.

Ranks x Node	MPI+OMP			TAMPI+OSS		
	Total	Refine	No Refine	Total	Refine	No Refine
1	485.2	43.2	442.0	469.8	19.8	450.0
2	375.4	39.5	335.9	303.9	23.5	280.4
4	352.0	36.0	316.0	306.2	21.6	284.6
8	348.6	30.5	318.1	314.5	18.3	296.2
16	344.0	29.1	314.9	322.3	19.4	302.9

The refinement with one rank per node is a special case. The nature of the single sphere input may produce even more imbalance when running with fewer ranks, so most of the pressure falls on the load balancing section. In this part, the TAMPI+OSS's parallelization performs better than the MPI+OMP's, but we will see that the difference is not relevant when scaling to more nodes.

B. Trace Analysis & Additional Options

We continue the evaluation by analyzing the execution traces of MPI-only and TAMPI+OSS using their optimal configuration. We extract traces using Extrae [28] and we visualize them on Paraver [29]. For graphical reasons, these executions should be relatively small in terms of nodes and time, so we have prepared a small input problem that mimics the one that we will use in the scaling analysis. This experiment simulates the four spheres problem on two nodes during nine timesteps. The mini-app will perform 20 stages per timestep, a refinement every five timesteps, and a checksum validation every ten stages. Mesh blocks are composed of 12x12x12 grid cells, each holding 20 variables. Lastly, we decrease the maximum refinement level of blocks to reduce the problem size.

Figure 1 shows the execution traces for MPI-only (upper) and TAMPI+OSS (lower) on the same time scale. The MPI-only has 96 ranks and the TAMPI+OSS has 8 ranks with 12 cores/rank. The former shows the MPI communication calls, while the latter shows the execution of OmpSs-2 tasks. Four different parts can be distinguished in both traces. The first part is the initial refinement phase, which is executed before starting the main loop of miniAMR. Refinement phases are shown in the upper trace as dense pink areas, which mainly correspond to collective operations. In the lower trace, in the same region, they predominate various colorful tasks that perform the block packing/unpacking, sending/receiving,

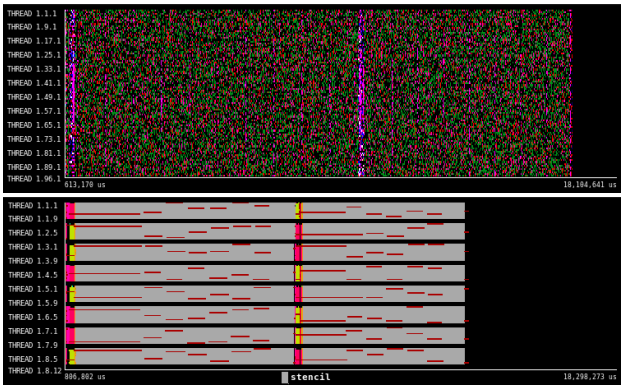

Fig. 1. Execution traces for MPI-only (upper) and TAMPI+OSS (lower) on 2 nodes at the same scale. The Y-axis shows MPI ranks/OmpSs-2 threads and the X-axis is the timeline.

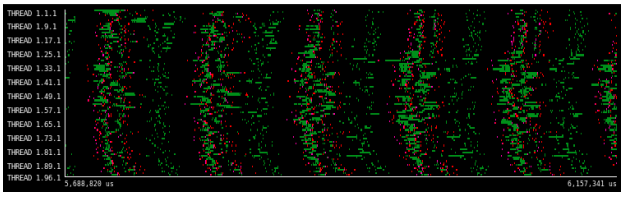

Fig. 2. Execution trace zooming in the stencil, communication and checksum phases for MPI-only on 2 nodes, showing several execution stages.

and splitting/coarsening, which are only present in refinement phases. The next region is the execution of the first five timesteps. Inside that region, there are stencil computations, intra- and inter-process communications, and checksums. In the TAMPI+OSS trace, stencil tasks are shown in gray. However, we will zoom into that region to see that more task types are being executed. Notice that the non-refinement region is significantly smaller in our variant, achieving an approximate speedup of 1.3x. Next, there is a second refinement, and then, the execution of the remaining four timesteps.

We focus on the non-refinement parts since they are the largest ones. However, in our experiments, the refinement time is usually around 8 and 10% of the total, while here is 3%. Figure 2 zooms into a non-refinement portion from the previous MPI-only trace. This figure shows the MPI calls in different colors, whereas the empty parts (in black) represent the regions without MPI (e.g., computations and control code). We clearly distinguish the execution of different loop stages, where colorful regions belong to communication phases. The green parts represent the calls to `MPI_Waitany` made in the `communicate` function. The red and pink dots represent the execution of `MPI_Isend` and `MPI_Irecv` operations, respectively, which are small due to their non-blocking nature. Lastly, the black parts are the block face packing/unpacking, the intra-process communication, and the stencil.

Similarly, the upper trace in Figure 3 zooms into a non-refinement part of the TAMPI+OSS trace. In the previous hybrid trace, we saw only stencil tasks in non-refinement areas due to graphical filtering. However, in these zoomed traces, we can see the execution of multiple types of tasks. Although the predominant are still the stencil tasks (gray), there are

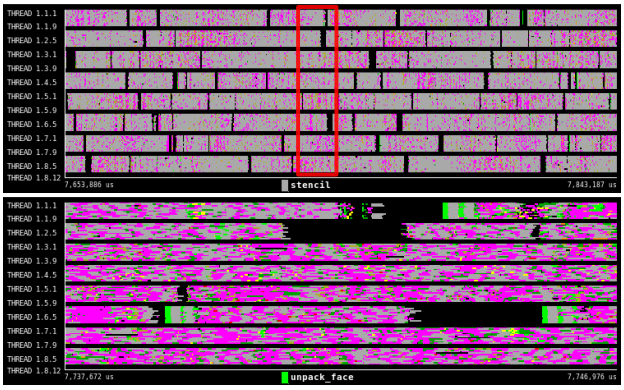

Fig. 3. Execution traces zooming in the stencil, communication and checksum phases for TAMPI+OSS on 2 nodes. The upper trace shows several execution stages, while the lower trace zooms in the red rectangular region.

also intra-communication (pink) and block face unpacking tasks (electric green). We hardly see MPI communication tasks because they call non-blocking MPI/TAMPI functions, so they are very small. In fact, MPI communications are transparently overlapped with the rest of tasks thanks to TAMPI.

The execution is very dense in the sense that almost all cores are running tasks constantly, and also, tasks from different phases are overlapping. However, there are a few spaces without parallelism. These empty regions take less than three milliseconds, and they are mainly produced because there are TAMPI communications in-flight waiting for data coming from other ranks. The lower trace zooms into the highlighted area in the upper trace. There, in the first rank, some unpacking tasks (in electric green) are executed immediately after the blank space. That means that some data has arrived, and the unpacking tasks can now execute. Once an unpacking task finishes, the next stencil task (in gray) can run to update the corresponding mesh block. Notice that this pattern is also repeated in the sixth rank.

Nevertheless, some blank spaces are not followed by unpacking tasks but by packing or intra-process communication tasks, e.g., the second rank. Although it needs further investigation, it seems that those spaces are produced by system noises. It is not a vital problem because they only take three milliseconds and they appear occasionally.

We have investigated the difference in performance between our approach and both MPI-only/MPI+OMP using Paraver. The four causes of our improvement are that (1) computation and communication phases are clearly overlapping, (2) communication tasks can be reordered, (3) we are less sensitive to load imbalance, and (4) we significantly improve the number of instructions per cycle (IPC). In our approach, tasks have lower cache miss ratios because the tasking runtime enables an immediate successor execution policy. Once a CPU finishes a given task, the same CPU starts executing its successor task such that it can reuse the data on the cache.

During this analysis, we have not seen any problem with the granularity of tasks, therefore, we will keep this configuration for the rest of experiments. Regarding the MPI+OpenMP execution, we have not added any execution trace because

TABLE II
THE NON-REFINEMENT TIME (S) VARYING THE NUMBER OF COMMUNICATION TASKS PER NEIGHBOR AND DIRECTION ON 64 NODES.

Tasks	1	2	4	8	16	all
Time(s)	612.5	600.0	594.9	595.5	597.8	627.5

there is no relevant information to show. The only aspects to highlight are that the execution time is almost the same as the MPI-only variant and that all communications are done sequentially from the main thread.

We have also executed our TAMPI+OSS variant on 64 nodes to find the most suitable value for the `--max_comm_tasks` option. It controls the number of communication tasks (and MPI messages) per neighbor and direction that are instantiated in the `communicate` phase, when `--send_faces` is enabled. In this experiment, we simulate the four spheres problem on 64 nodes. We show the total execution time of all non-refinement stages in Table II. The refinement time is not considered because that option should not affect that part. Firstly, we have executed from 1 to 16 communication tasks per neighbor and direction. Then, we have executed the program without limiting the parallelism, instantiating a task per block face to be transmitted. Notice that this last configuration, named as *all* in Table II, exposes all parallelism in that phase. The range of best values for that option is from 4 to 16. From now on, we use eight communication tasks per neighbor and direction, since it is the value that offers both reasonable performance and parallelism. Note that none of these options apply to MPI-only and MPI+OMP variants.

C. Weak Scaling

Our next experiment is to evaluate the scalability of the three variants up to 256 nodes, simulating the four spheres problem. The execution consists of 99 timesteps and 40 stages per timestep, mesh blocks of 12x12x12 grid cells, and cells with 40 variables. The refinement stage is performed every five timesteps and the checksum validation every ten stages. We also limit to one the refinement levels that a mesh block can be refined during a refinement stage. For TAMPI+OSS, we use `--separate_buffers` and `--send_faces`, we limit the communication tasks per neighbor/direction to eight, and we enable the checksum optimization.

The idea is to double the problem size when doubling the number of nodes, performing a weak scaling analysis. Note that each variant uses its optimal configuration of ranks per node, so that they will leverage a distinct total number of ranks. We design the experiment focusing on the constraint that all variants must have the same initial mesh. Thus, the total number of initial blocks in all variants is the same. Also, each variant may have different number of ranks in a particular direction, but they all have the same total number of blocks in that direction. These constraints guarantee the same conditions such that we can fairly compare them. Notice that the most restrictive configuration is MPI-only, which has 48 ranks per node. Thus, we decided to use one initial block per rank in MPI-only, and arrange the ranks in the three directions as uniformly as possible. We mimic the same block arrangement

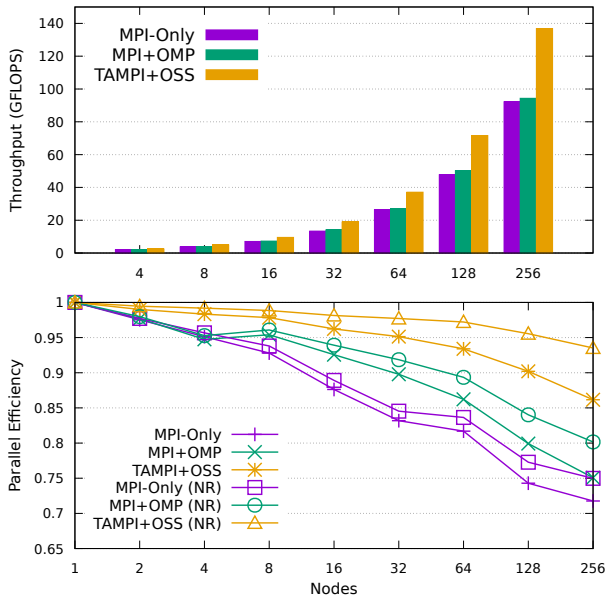

Fig. 4. The weak scaling's throughput (upper) from 4 to 256 nodes and the efficiency (lower) from 1 to 256 nodes. The lower shows the efficiency for both the total time and assuming a negligible refinement time (NR).

in hybrid variants by placing more initial blocks per rank and direction. In this way, all have the same mesh shape and size while maintaining a fixed block size. Then, when doubling the number of nodes, we double the number of total blocks in one of the directions following a round-robin fashion.

Figure 4 (upper) shows the throughput in GFLOPS from 4 to 256 nodes for all three versions. We compute the throughput dividing the total number of floating-point operations by the total execution time. The number of operations is already reported by the mini-app and counts the total operations performed during the stencil phases. As we can see, our TAMPI+OSS approach is getting the best performance in all cases. For instance, we obtain a 1.50x and 1.49x throughput speedup w.r.t. MPI-only on 128 and 256 nodes, respectively.

Our refinement takes 8.1% of the total execution time on 256 nodes. Our speedup in the non-refinement part reaches its maximum value 1.54x on 256 nodes, following an increasing tendency in the speedup as we increase the number of nodes. In this figure, we only show the best configuration for our approach, which is four ranks/node. The two ranks/node configuration has a similar throughput in non-refinement stages, but the refinement time increases significantly. This deterioration is because the refinement is not fully parallelized, and the work per rank with four ranks/node is smaller than with two. As an example, the refinement time with two ranks/node takes 1.31x more time than the time with four. In contrast, the MPI+OMP fork-join, also with four ranks/node, does not exceed the 1.06x speedup w.r.t. MPI-only, and only achieves a 1.06x speedup in the non-refinement phases. Moreover, it does not reach the MPI-only throughput from one to four nodes.

Finally, we plot their parallel efficiency in the weak scaling experiment in Figure 4 (lower). This figure helps understanding how efficiently each of the variants scale. The efficiency

is computed with respect to each variant's throughput in one node. We show the efficiency of the whole execution time, but also the efficiency without considering the refinement time (marked as NR). Conceptually, the NR efficiency is the one we could see in an ideal execution where the refinement would take negligible time to run. That allows us to observe the impact of the refinement phases. Again, our approach is the one that best scales across all executions. Our efficiency starts to decrease steadily on eight nodes and ends with a 0.86 efficiency on 256 nodes. Our non-refinement efficiency is significantly better and reaches 0.94 on 256 nodes, which is relatively high when leveraging this amount of resources. Conversely, the efficiency for MPI-only and MPI+OMP on 256 nodes reaches 0.72 and 0.75, respectively. As expected, their non-refinement efficiency is significantly better.

D. Strong Scaling

Our last experiment evaluates the strong scalability of the three variants, keeping a constant problem size. We simulate the four spheres problem during 79 timesteps, 40 stages per timestep, and mesh blocks of 10x10x10 40-variable grid cells. The total number of blocks per direction is the same as in the weak scaling experiment on 256 nodes. However, the resource requirements (computation time and memory) of that input were too high when running with 1 to 8 nodes, so we have fairly divided it by 16 in those cases.

Figure 5 shows the throughput's speedup (upper) and the efficiency (lower) when varying the number of nodes. We compute the speedup with respect to the throughput of the MPI-only variant in one node and taking into account the whole execution time. The efficiency is computed with respect to each variant's throughput in one node. We only show the four ranks/node configuration for the hybrid variants. Again, our TAMPI+OSS approach is the one performing and scaling better in all cases. On 256 nodes, we still have a 0.88 overall efficiency, and we are 1.60x faster than MPI-only.

In contrast, the fork-join MPI+OMP variant is slightly better than MPI-only from 8 to 128 nodes, but its efficiency drops faster as we increase the amount of nodes, so it ends performing worse on 256 nodes. Notice both MPI-only and MPI+OMP have similar behavior in the efficiency. It decreases initially, then it stabilizes when using 8 to 32 nodes, and finally, it decreases again from 64 nodes. Note that the MPI-only version stops the drop of efficiency on 256 nodes, and thus, it exceeds the MPI+OMP's throughput. These efficiency drops are not observed in TAMPI+OSS, which smoothly decreases during the experiment, meaning that our approach is less sensitive to the increase of communications and neighbors.

VI. Conclusions and Future Work

In this paper, we have fully taskified the miniAMR proxy application, including the computation, communication, refinement, and load balancing phases. We describe the steps and modifications required to go from an MPI-only version to a hybrid one using the OmpSs-2 programming model and the TAMPI library, which are straightforward to apply in most

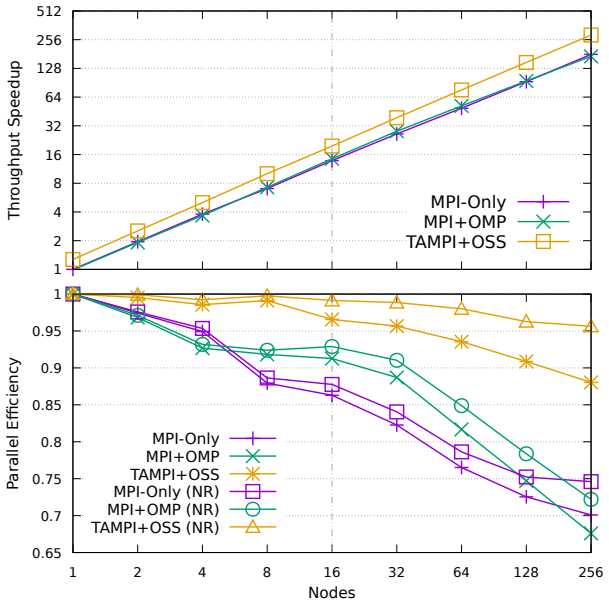

Fig. 5. The strong scaling's speedup (upper) and the efficiency (lower) from 1 to 256 nodes. The lower shows the efficiency for both the total time and assuming a negligible refinement time (NR). Due to the memory available in each node, we use a large input for the experiments from 16 to 256 nodes, and a 16x smaller input for the experiments from 1 to 8 nodes.

cases. Our in-depth evaluation shows the benefits of a pure data-flow execution model when running AMR applications at scale. We have evaluated our approach comparing it to the reference MPI-only version and a hybrid MPI+OpenMP fork-join version. On a weak scaling with up to 12288 cores, we observed that our hybrid variant achieves a speedup of 1.5x w.r.t. MPI-only, while the fork-join does not exceed 1.06x. That improvement is due to the data-flow execution model of our hybrid version, which can effectively overlap application phases, naturally reorder communication tasks, and significantly increase the IPC of computation tasks. The increase in the IPC is due to the OmpSs-2 scheduling heuristics that leverage the task-graph to exploit temporal locality. We also show that by exposing application parallelism through annotations, a data-flow execution model can significantly improve application performance without sacrificing programmability.

As future work, we plan to investigate how other challenging applications can benefit from a pure data-flow execution model when running at scale, and how we could integrate that model into AMReX/BoxLib applications.

Acknowledgments

We thank Dr. Courtenay T. Vaughan and Dr. Clay Hughes (Sandia National Laboratories, Albuquerque, NM, USA) for assisting us in the process of understanding the details of miniAMR and giving advice on relevant input problems for our experiments. This work has been supported by the European Union H2020 Programme through the DEEP-EST project (agreement No. 754304), the Spanish Government through the Severo Ochoa Program (SEV-2015-0493), the Spanish Ministry of Science and Innovation (PID2019-107255GB), and the Generalitat de Catalunya (2017-SGR-1414).

REFERENCES

[1] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard. Version 3.1*. University of Tennessee, Jun. 2015.

[2] R. Rabenseifner and G. Wellein, "Comparison of Parallel Programming Models on Clusters of SMP Nodes," in *Proceedings of the International Conference on High Performance Scientifc Computing*, Mar. 2003.

[3] R. Rabenseifner, "Hybrid parallel programming: Performance problems and chances," in *45th Cray User Group Conference*, 2003, pp. 12–16.

[4] G. Jost, H. Jin, and F. F. Hatay, "Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster," NASA, Tech. Rep., Sep. 2003.

[5] OpenMP Architecture Review Board, *OpenMP Application Programming Interface: Version 5.0*, Nov. 2018, available at https://www.openmp.org/specifications.

[6] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig *et al.*, "The international exascale software project roadmap," *The international journal of high performance computing applications*, vol. 25, no. 1, pp. 3–60, 2011.

[7] A. Amer, H. Lu, P. Balaji, and S. Matsuoka, "Characterizing MPI and hybrid MPI+ Threads applications at scale: case study with BFS," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2015, pp. 1075–1083.

[8] K. Sala, X. Teruel, J. M. Perez, A. J. Peña, V. Beltran, and J. Labarta, "Integrating blocking and non-blocking MPI primitives with task-based programming models," *Parallel Computing*, vol. 85, pp. 153–166, 2019.

[9] Barcelona Supercomputing Center, "Task-Aware MPI (TAMPI) Library." [Online]. Available: https://github.com/bsc-pm/tampi

[10] ——, "OmpSs-2 Specification." [Online]. Available: https://pm.bsc.es/ompss-2-docs/spec/

[11] J. Ciesko, P. J. Martínez-Ferrer, R. P. Veigas, X. Teruel, and V. Beltran, "HDOT—An approach towards productive programming of hybrid applications," *Journal of Parallel and Distributed Computing*, vol. 137, pp. 104–118, 2020.

[12] M. J. Berger, P. Colella *et al.*, "Local adaptive mesh refinement for shock hydrodynamics," *Journal of computational Physics*, vol. 82, no. 1, pp. 64–84, 1989.

[13] C. T. Vaughan and R. F. Barrett, "Enabling tractable exploration of the performance of adaptive mesh refinement," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 746–752.

[14] C. T. Vaughan, "MiniAMR Adaptive Mesh Refinement (AMR) Mini-app." [Online]. Available: https://github.com/Mantevo/miniAMR

[15] J. McGlaun, S. Thompson, and M. Elrick, "CTH: A three-dimensional shock wave physics code," *International Journal of Impact Engineering*, vol. 10, no. 1, pp. 351 – 360, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0734743X90900713

[16] A. Rico, I. S. Barrera, J. A. Joao, J. Randall, M. Casas, and M. Moretó, "On the benefits of tasking with OpenMP," in *International Workshop on OpenMP*. Springer, 2019, pp. 217–230.

[17] Mantevo Organization, "Mantevo Project." [Online]. Available: https://mantevo.github.io

[18] Barcelona Supercomputing Center, "OmpSs Specification." [Online]. Available: https://pm.bsc.es/ompss-docs/specs/

[19] D. S. Balsara and C. D. Norton, "Highly parallel structured adaptive mesh refinement using parallel language-based approaches," *Parallel Computing*, vol. 27, no. 1-2, pp. 37–70, 2001.

[20] H.-Y. Schive, U.-H. Zhang, and T. Chiueh, "Directionally unsplit hydrodynamic schemes with hybrid MPI/OpenMP/GPU parallelization in AMR," *The International Journal of High Performance Computing Applications*, vol. 26, no. 4, pp. 367–377, 2012.

[21] P. Samfass, J. Klinkenberg, and M. Bader, "Hybrid MPI+ OpenMP Reactive Work Stealing in Distributed Memory in the PDE Framework sam (oa)ˆ 2," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 337–347.

[22] R. Prat, L. Colombet, and R. Namyst, "Combining task-based parallelism and adaptive mesh refinement techniques in molecular dynamics simulations," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–10.

[23] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, M. Katz, A. Myers, T. Nguyen, A. Nonaka, M. Rosso, S. Williams, and M. Zingale, "AMReX: a framework for block-structured adaptive mesh refinement," *Journal of Open Source Software*, vol. 4, no. 37, p. 1370, May 2019. [Online]. Available: https://doi.org/10.21105/joss.01370

[24] J. Bell, A. Almgren, V. Beckner, M. Day, M. Lijewski, A. Nonaka, and W. Zhang, "Boxlib user's guide," *github. com/BoxLib-Codes/BoxLib*, 2012.

[25] P. Colella, D. T. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. Van Straalen, "Chombo software package for amr applications design document," *Available at the Chombo website: http://seesar. lbl. gov/ANAG/chombo/(September 2008)*, 2009.

[26] W. Zhang, A. Almgren, M. Day, T. Nguyen, J. Shalf, and D. Unat, "Boxlib with tiling: An adaptive mesh refinement software framework," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S156–S172, 2016.

[27] M. N. Farooqi, T. Nguyen, W. Zhang, A. S. Almgren, J. Shalf, and D. Unat, "Phase asynchronous amr execution for productive and performant astrophysical flows," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 880–893.

[28] Barcelona Supercomputing Center, "Extrae." [Online]. Available: https://tools.bsc.es/extrae

[29] ——, "Paraver." [Online]. Available: https://tools.bsc.es/paraver