# On the Benefits of Tasking with OpenMP

Alejandro Rico[1], Isaac Sánchez Barrera[2,3], Jose A. Joao[1], Joshua Randall[1], Marc Casas[2], and Miquel Moretó[2,3]

[1] Arm Research, Austin, TX, USA
{alejandro.rico,jose.joao,joshua.randall}@arm.com
[2] Barcelona Supercomputing Center, Barcelona, Spain
{isaac.sanchez,marc.casas,miquel.moreto}@bsc.es
[3] Universitat Politècnica de Catalunya, Barcelona, Spain

**Abstract.** Tasking promises a model to program parallel applications that provides intuitive semantics. In the case of tasks with dependences, it also promises better load balancing by removing global synchronizations (barriers), and potential for improved locality. Still, the adoption of tasking in production HPC codes has been slow. Despite OpenMP supporting tasks, most codes rely on worksharing-loop constructs alongside MPI primitives. This paper provides insights on the benefits of tasking over the worksharing-loop model by reporting on the experience of taskifying an adaptive mesh refinement proxy application: miniAMR. The performance evaluation shows the taskified implementation being 15–30% faster than the loop-parallel one for certain thread counts across four systems, three architectures and four compilers thanks to better load balancing and system utilization. Dynamic scheduling of loops narrows the gap but still falls short of tasking due to serial sections between loops. Locality improvements are incidental due to the lack of locality-aware scheduling. Overall, the introduction of asynchrony with tasking lives up to its promises, provided that programmers parallelize beyond individual loops and across application phases.

**Keywords:** Tasking · OpenMP · Parallelism · Scaling

## 1 Introduction

Tasking is an important feature of multiple parallel programming models targeting both shared and distributed memory, such as Thread Building Blocks (TBB), Chapel, OmpSs, OpenACC, Kokkos, among others. OpenMP, mainstream programming model in the high performance computing (HPC) space, includes tasking since version 3.0 (2008) [2, 5] and tasking with dependences since version 4.0 (2013) through `task` constructs [6, 17, 18]. OpenMP also supports tasking for distributed memory with `target` constructs. Tasking is widely used to offload computation to accelerators in heterogeneous systems. CUDA, OpenCL and OpenACC kernels, and OpenMP target concepts are examples of this. However, the adoption of tasking for shared memory (threading) has been slow. Many

HPC codes include threading with OpenMP alongside MPI, mostly through the use of worksharing-loop constructs with fork-join semantics. For more developers to taskify their codes, the effort required and the resulting benefits need to be considered.

This paper is an assessment of the benefits promised by tasking. These benefits include an intuitive parallel work unit —a task— which can be defined as a piece of computation on a piece of data that could be run in parallel. They also include the ability to define data-flow semantics between tasks using dependences and remove expensive global synchronizations and their potential load imbalance. We contribute to the discussion on tasking adoption in the community with our experience taskifying an adaptive mesh refinement (AMR) proxy-app: miniAMR [12, 16]. This proxy-app It is part of the Mantevo [10] project and the Exascale Computing Project Proxy Apps Suite [7] and models the refinement and communication phases of AMR codes. It is programmed in MPI and OpenMP, the OpenMP parallelization using worksharing-loop constructs only. Our taskification focuses on removing global synchronization between communication and computation phases to reduce the inherent load imbalance of working on blocks at different refinement levels. A previous paper [14] improves miniAMR load imbalance at the MPI level by changing its algorithmic implementation. In this work, we focus on maintaining the algorithmic properties of the reference miniAMR implementation and replacing loop-level parallel regions by task regions. The goal is to quantify the resulting performance benefits and report on our experience to give guidance on how to taskify such type of parallel work and give a sense of the effort required.

We report better performance using tasks on multiple systems including Marvell ThunderX2, IBM POWER9, Intel Skylake-SP and AMD EPYC. Overall, the taskification experience shows that developers need to think on parallel work across application phases, which involves larger code sections than only focusing on individual loops. The results show that tasking provides 15-30% better performance for certain thread counts and across the evaluated platforms. These improvements are mainly due to removal of load imbalance and avoidance of serial sections leading to a higher thread utilization.

## 2 The miniAMR Proxy Application

Adaptive mesh refinement (AMR) was developed as a way to model the physical domain with different levels of precision in numerical problems [3, 4], with the goal of achieving higher precision in regions where it is needed (such as boundaries, points of discontinuity or steep gradients [4]). The physical domain is a rectangle (a rectangular prism in 3D space) that is represented as nested rectangular grids that share boundaries, with denser (finer) grids where higher precision is required.

The numerical algorithm is applied to each of the rectangles of the grid, with the corresponding communication on the boundaries between grid elements. The grid is updated when the conditions of the domain change: an error formula is defined to force the use of a finer grid when a threshold value is reached. The

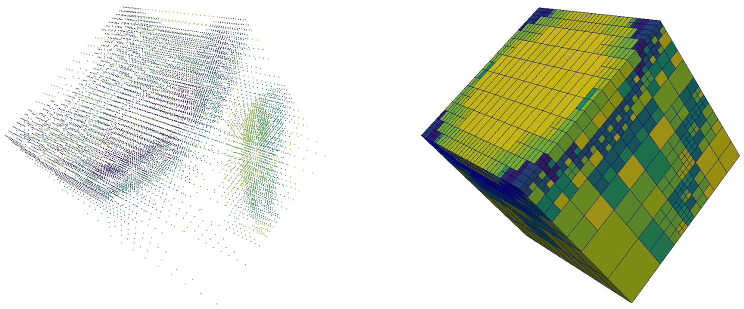

**Fig. 1.** Visualization of a unit cube with a domain defined by two empty spheres, using the vertices (left) and boundaries (right) of the grids. Colors have no special meaning.

refinement is carried out by splitting the elements of the grid into two equal parts in all dimensions. This means that, in 2D, each rectangle is split into 4 other rectangles (quadrants) and, in 3D, each prism is split into 8 prisms (octants).

MiniAMR is a proxy application released as part of version 3.0 of the Mantevo suite [10, 11] that is used to model the refinement/coarsening and communication routines of parallel AMR applications using MPI. The physical domain is modelled as a unit cube in 3D space divided in blocks in all three dimensions, which define the coarsest level of the grid.

To simulate the changes in the domain, miniAMR provides up to 16 different types of objects (both solid and surfaces), which include spheroids, cylinders, rectangles and planes. These objects can interact with the domain in different ways: moving at a constant speed, bouncing on the boundaries of the outside prism and growing on the $X$, $Y$ or $Z$ directions. Their positions determine the regions of the domain that need more precision and, therefore, a finer grid.

To simplify the communications, miniAMR forces neighboring blocks to be at distance 1 in the refinement level. This means that every face of a 3D block is a neighbor of a whole face (at the same refinement level), four other faces (which are finer) or a quarter of another face (which is coarser). A sample domain at a given time step can be seen in Fig. 1. All these blocks occupy the same bytes in memory; when refinement happens for a block, the resolution is doubled in each dimension by replacing that block by 8 new blocks.

The sample computations are modeled using different stencil algorithms, applying them to the different variables that are defined. For simplicity, we will focus on the 7-point stencil, where each discrete point is the average of itself and its six neighbor points in 3D space (up, down, north, south, east, west).

## 2.1 Baseline Parallelization of the miniAMR Code

To understand the changes to the code for taskification in Section 3, we first introduce how the application works originally according to the source code available in the Mantevo repository [12].

The initial, coarsest grid is given by the number of MPI ranks in each dimension and the number of initial blocks (grid cells) per MPI rank per dimension. The

**Algorithm 1:** miniAMR main loop

```
foreach time step or simulation time finished do
    foreach stage in time step do
        foreach communication group do
            communicate;
            foreach variable in communication group do
                stencil;
                if time for checksum then
                    checksum;
                    validate checksum;
                end
            end
        end
    end
    if time for refinement then
        refine;
    end
end
```

application does an initial allocation for all the blocks that can be used (limited by a user-specified parameter). In the original code, this is implemented as an array of structs, where each block struct contains a quadruple pointer to `double` (i.e., `double****`) with the first indirection for the total amount of variables, one indirection per dimension, and memory contiguity only in the $Z$ axis. Each dimension has two extra elements to allow for an extra face on each side of the block to account for *ghost values* (as the values in the boundaries of neighbor blocks are called in the miniAMR code). Blocks that are not in use are marked as such so that they can be used in future refinements.

Algorithm 1 shows the pseudo-code of the main loop that is executed after initialization. The main loop runs for a total number of time steps or a given simulation time. This loop is divided in stages that start with the communications between neighboring cells followed by the stencil updates, sometimes followed by a checksum calculation. These pairs of communication and stencil are grouped by a certain number of variables (communication group). For example, the total number of variables is 40, while communications and stencil updates are done in groups of 10 variables. Every few stages, the objects in the domain are moved according to the parameters, the domain is refined/coarsened following the settings, and the main loop starts again.

The communications are done for both local (intraprocess) and external (interprocess) neighboring blocks, MPI non-blocking calls being used for the second case. When the blocks are of the same size, the ghost values are simply copied. If a face has four neighbors, because the neighbor grid is finer, the values are replicated four times and the variables are divided by 4 to keep the total value constant. Similarly, all ghost values received by the coarser face are added up in groups of four to a single discrete point.

**Table 1.** MiniAMR versions developed in this work

| Label | Description |
|---|---|
| Orig | Original code from Mantevo repository with stencil parallel loop fixed |
| Orig-dyn | Orig with dynamically scheduled comm |
| Loop | Transformation of main data structure into contiguous array |
| Loop-dyn | Loop with dynamically scheduled comm |
| Task-1 | Data-flow parallelization of comm and stencil. Taskloop for checksum |
| Task-2 | Data-flow parallelization of comm, stencil and checksum |

When splitting a block in the refinement process, each original point is replicated 8 times and its variables are divided by 8 in order to preserve the total value, as when communicating. The coarsening process is equivalent: 8 blocks are joined to form a coarser block, so the points are added up in groups of 8 to form a coarser point.

## 3 Taskification of MiniAMR

Table 1 lists the versions developed in this work towards the taskification of miniAMR using OpenMP. The parallelization of miniAMR in the reference code of the Mantevo project is based on MPI and OpenMP. Message passing between processes occurs mainly in the communication phase when the faces of blocks (ghost values) are transferred in a process commonly known as halo exchange. An `MPI_AllReduce` primitive coordinates all processes to calculate the overall checksum. MPI is also used in other parts of the code outside of the main phases that are outside of the scope of this analysis, such as a plotting phase to visualize the simulated grid like the one shown in Fig. 1. OpenMP is used in the communication phase to exchange halos between threads, the computation phase (stencil) and checksum calculation. The refinement phase is serial.

The first transformation of the code (labeled as Orig) is to correct the original stencil OpenMP parallelization, which gives incorrect results as of February 14th, 2019 (the latest commit in the master branch at the time of writing). This issue was communicated to miniAMR developers. Listing 1 shows the resulting OpenMP annotation on the 7-point stencil code.

The taskification strategy is that a task communicates (`comm`) or computes (`stencil`) the variables of one block. It is beneficial for the data belonging to the variables of a block to be contiguous in memory so task dependencies can be expressed as array sections. To prepare the code towards taskfication, the second transformation is to change the main data structure from a quadruple pointer (`double****`) with disaggregated arrays for each block, variable, and $X$, $Y$ and $Z$ dimensions, into a contiguous array (`double*`). This version (labeled as Loop) is our reference loop-parallel version using worksharing-loop constructs only. Having a contiguous array improves performance over the original code thanks to better prefetching coverage and accuracy due to improved locality. To isolate this improvement from that provided by taskification, the performance results in Section 5 are normalized to Loop.

```
#pragma omp parallel for default(shared)
//loop over blocks
for (int in = 0; in < sorted_index[num_refine+1]; in++) {
  block *bp = &blocks[sorted_list[in].n];
  block3D_t array = (block3D_t)&bp->array[var*block3D_size];
  double work[x_block_size+2][y_block_size+2][z_block_size+2];
  memcpy(work, array, sizeof(work)); //save in temp storage
  for (int i = 1; i <= x_block_size; i++)
    for (int j = 1; j <= y_block_size; j++)
      for (int k = 1; k <= z_block_size; k++)
        array[i][j][k] = (work[i-1][j  ][k  ] +
                          work[i  ][j-1][k  ] +
                          work[i  ][j  ][k-1] +
                          work[i  ][j  ][k  ] +
                          work[i  ][j  ][k+1] +
                          work[i  ][j+1][k  ] +
                          work[i+1][j  ][k  ])/7.0;
  }
```

**Listing 1.** 7-point stencil code with correct worksharing construct.

```
double *barray  = bp->array;
double *barray1 = bp1->array;
#pragma omp task \
    depend(inout: barray[start*bsize:num_comm*bsize], \
                  barray1[start*bsize:num_comm*bsize]) \
    firstprivate(...) default(none)
{
   //loop over variables in communication group
   for (int m = start; m < start+num_comm; m++) {
      block3D_t array  = (block3D_t)&barray[m*bsize];
      block3D_t array1 = (block3D_t)&barray1[m*bsize];
      //exchange face ghost values
      for (int j = 1; j <= y_block_size; j++)
         for (int k = 1; k <= z_block_size; k++) {
            array1[x_block_size+1][j][k] = array[1][j][k];
            array[0][j][k] = array1[x_block_size][j][k];
         }
   }
}
```

**Listing 2.** Communication task between blocks at the same refinement level. `bp` and `bp1` are pointers to the blocks exchanging faces. `bsize` is the 3D block size. Blocks are laid out contiguously for each variable

The third version (labeled Task-1) is the taskification of the communication, stencil and checksum phases on top of Loop. In the original code, the loop in the communication phase traverses all blocks and each iteration performs ghost value exchanges between a block face and a neighbor face at the same or different refinement level. This loop is distributed across threads with an `omp parallel for` construct. In this taskification, this worksharing-loop construct is removed and a task is defined for each exchange inside the loop. Listing 2 shows the task code for a face exchange at same refinement level. Tasks read and write to a part of the block and the dependence is set for the whole block. This could be improved by

```
#pragma omp taskwait
//original: #pragma omp parallel for reduction(+: sum)
#pragma omp taskloop
for (int in = 0; in < sorted_index[num_refine+1]; in++) {
  block *bp = &blocks[sorted_list[in].n];
  double block_sum = 0.0;
  block3D_t array = (block3D_t)&bp->array[var*bsize];
  for (int i = 1; i <= x_block_size; i++)
     for (int j = 1; j <= y_block_size; j++)
        for (int k = 1; k <= z_block_size; k++)
           block_sum += array[i][j][k];
  //update check sum
#pragma omp atomic
  sum += block_sum;
}
```

**Listing 3.** Checksum task for Task-1 using taskloop

arranging halos with ghost values in separate arrays and having dependences only on halos instead, or by adding a separate dependence for each halo and variable. However, both of these solutions add complexity either to the data structure or to the directive readability, so this is not included in the version evaluated here. We expect support for multidependences [8, 17] in OpenMP 5.0 to help with the directive readability issue (we must restrict this effort to OpenMP 4.5 features due to current compiler support). Stencil computations are taskified with an `inout` dependence on the block they operate on, and therefore depend on the previous communication tasks that write to that block. With this data-flow dependence strategy, a pair of `parallel` and `single` directives surround the loop iterating over the stages in the main loop, therefore removing the implicit barrier between the communication and stencil phases that worksharing-loop constructs in the original code imply.

At this point there is data flow between communication and stencil computation. Due to being inside a `parallel-single` pair, the worksharing-loop construct around checksum executes serially on one thread. Given that checksum does not execute on every iteration, this taskification uses a `taskloop` construct [15], which executes the iterations of checksum over the blocks in tasks, and therefore has the same implicit barrier after the checksum loop as the previous worksharing-loop construct. Listing 3 shows the corresponding task code. To make sure prior tasks complete before checksum, a `taskwait` primitive is placed before the checksum task loop. The refinement phase is outside of the task region and therefore remains serial as in the original code. Taskifying the refinement phase to overlap iterations across timesteps is a potential improvement left for future work.

The fourth version (labeled as Task-2) builds on top of Task-1 and replaces the `taskloop`-based taskification of checksum by data-flow using dependencies. Listing 4 shows the task code. The loop iterating over the variables in the block is brought inside the task and the partial checksum variable becomes an array with an entry for each variable. This removes the `taskwait` before the checksum phase and allows hoisting the checksum for a given block as soon as its stencil

```
for (int in = 0; in < sorted_index[num_refine+1]; in++) {
  block *bp = &blocks[sorted_list[in].n];
  double *barray = bp->array;
#pragma omp task \
  depend(in: barray[var*bsize:number*bsize]) \
  firstprivate(...) default(none)
  {
    //loop over variables in communication group
    for (int v = var; v < var+number; ++v) {
      block3D_t array = (block3D_t)&barray[v*bsize];
      double block_sum = 0.0;
      for (int i = 1; i <= x_block_size; i++)
         for (int j = 1; j <= y_block_size; j++)
            for (int k = 1; k <= z_block_size; k++)
               block_sum += array[i][j][k];
      //update check sum for a given variable
#pragma omp atomic
      sum[v] += block_sum;
    }
  }
}
#pragma omp taskwait
```

**Listing 4.** Checksum task for Task-2 using data-flow dependences

is complete. The `taskwait` moves down after the creation of checksum tasks so checksum validation happens once all checksum tasks are complete.

Given the intrinsic load imbalance of the communication phase due to different block communications happening at different refinement levels, Table 1 includes two more versions of the code. Orig-dyn and Loop-dyn use dynamic scheduling by adding the clause `schedule(dynamic)` to the parallel loop in the communication phase to mitigate this imbalance and have another point of comparison between statically-scheduled loops and tasking.

This effort covers the shared memory portion of the application by replacing loop-level parallelization of communication, stencil and checksum with task-level parallelization to compare both models. The taskification of the MPI part promises further improvements given that it already uses asynchronous message passing. The evaluation of MPI communication tasking is left as future work.

## 4 Experimental Methodology

The experiments focus on comparing the worksharing-loop parallel and task-based implementations of miniAMR described in Section 3. As in prior work [1], they are run on multiple systems with different architectural and microarchitectural features and using different OpenMP C/C++ compiler and runtime systems to quantify the sensitivity to the underlying system features and runtime implementation. Table 2 shows the testbed systems and compilers used in this work.

We run miniAMR with multiple variations of input parameters that affect different parts of the application. We test multiple block sizes and number of

**Table 2.** Systems used for evaluation

| | | | | |
|---|---|---|---|---|
| | | SYSTEM | | |
| Name | Marvell ThunderX2 | IBM POWER9 | Intel Skylake-SP | AMD EPYC |
| Part no. | CN9975 | 8335-GTH | Xeon Platinum 8160 | 7401P |
| Processors | 2 | 2 | 2 | 1 |
| Memory | 16xDDR4-2666 | 16xDDR4-2666 | 12xDDR4-2666 | 8xDDR4-2666 |
| | | PROCESSOR | | |
| Cores | 28 | 20 | 24 | 24 |
| L1D cache | 32KB/core | 32KB/core | 32KB/core | 32KB/core |
| L2 cache | 256KB/core | 512KB/2 cores | 1MB/core | 512KB/core |
| L3 cache | 32MB | 120MB | 33MB | 64MB |
| NoC | Ring | - | Mesh | 4-die MCM |
| | | SOFTWARE | | |
| Compilers | GNU-8.2<br>Arm 19.1 | GNU-8.1<br>IBM XL 16.1 | GNU-8.1<br>Intel 19.0 | GNU-8.2 |

variables, which directly affect parallel work duration - often a performance limiting factor [9, 13]. The default block size in miniAMR is $10 \times 10 \times 10$ and previous papers used $64 \times 64 \times 64$ [14]. We use $16 \times 16 \times 16$ as a reasonable input and $8 \times 8 \times 8$ as a deliberately small block size to stress tasking overheads. The default number of variables is 40. We use 40 and 160 as a deliberately large input to isolate tasking overheads. We test checksum frequencies of one every five, and one every ten stages, which affects tasking look ahead as checksum validation implies a barrier. We test 10 and 40 stages per time step which affects refinement frequency —more stages per time step means less relative time spent in the refinement phase. The number of overall refinements is 4, maximum blocks is 3000 and simulation starts with 1 block. The simulated object, position, direction and speed is defined with parameters: `-num_objects 1 -object 2 0 -1.1 -1.1 -1.1 0.060 0.060 0.060 1.1 1.1 1.1 0.0 0.0 0.0`. The memory footprint of these runs is between 900MB and 20GB.

Experiments compare the execution time of the multiple variants (lower is better) varying the number of OpenMP threads in one MPI rank. The execution time of each phase is measurable only for the worksharing-loop versions, and therefore not relevant in this study because when global synchronizations are removed the execution of multiple phases overlap. The executions are done multiple times to mitigate variation across runs. Most systems show a small variation between runs, so one of them is shown here except for EPYC. This system showed the largest variation, so experiments were run 10 times and the results shown are the average after removing outliers ($\pm 2\times$ standard deviation).

## 5 Performance Evaluation

Figure 2 shows the normalized execution time (lower is better) of the multiple implementations of miniAMR, each subplot corresponding to a different platform, and each cluster of bars being for a different number of threads. All results

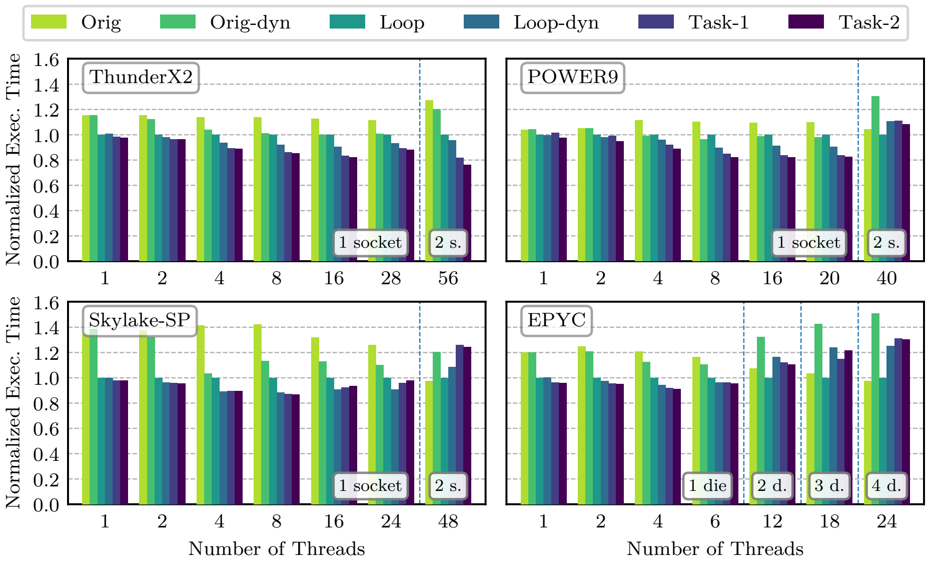

**Fig. 2.** Execution time of multiple miniAMR implementations on testbed systems; normalized to Loop

are using the GNU compiler and normalized to the Loop implementation. The parameters for this execution are: checksum frequency is every 5 stages, number of refinements is 4, blocks are 16×16×16, with 40 variables and 40 stages per timestep. We focus on this configuration as it is a representative input after discussion with application developers. A discussion of the performance variations of sweeping parameters is included later in this section.

In all cases, Loop is faster than the original version of the code (Orig) because of improved locality while accessing the main data structure, which is a contiguous array instead of being segregated per dimension. The two task implementations are generally better than the Loop version due to load imbalance mitigation in the communication phase and, for the larger core counts, also the stencil phase. Loop-dyn also improves over Loop due to better load balancing and outperforms tasking in some cases. However, in most cases, tasking is superior to dynamically-scheduled loops due to the serial portion in between parallel loops becoming increasingly important with increasing thread counts (Amdahl's Law).

When crossing socket or die boundaries (e.g., 56 cores in ThunderX2 are in two sockets, see Table 2), the dynamically-scheduled configurations (Orig-dyn, Loop-dyn, Task-1 and Task-2) show worse performance than statically-scheduled ones (Orig and Loop) in most cases. This is due to a large drop in performance of execution of both stencil and communications due to NUMA/NUCA effects. Static scheduling suffers heavily from load imbalance at the large core counts tested across sockets but has better caching behavior due to the same blocks being processed in the same threads across stages. With dynamic scheduling, each block is processed in potentially different threads across stages. The result is that the

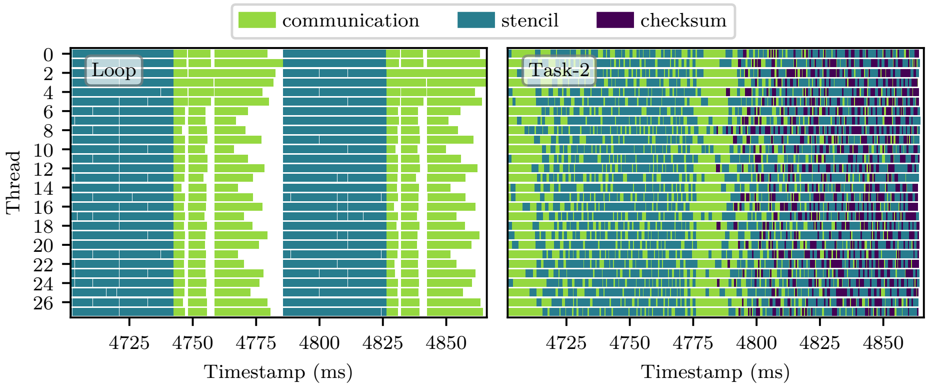

**Fig. 3.** Execution timelines of Loop (left) and Task-2 (right). White color is idle time

drop in instructions per cycle (IPC) on each thread for static scheduling is smaller than for dynamic scheduling when going from one socket to two sockets. In the case of EPYC, this is noticeable already at 12 threads because only 6 threads are co-located within the same die so over 6 threads is already a cross-chiplet execution paying larger NUMA latencies. Given the lack of performance counters that measure accesses to remote NUMA domains in the evaluated platforms, we plan to further analyze the impact of cross-socket/cross-chiplet accesses using simulated platforms in future work.

Figure 3 shows a timeline of the Loop (left) and Task-2 (right) versions showing execution of parallel loops and tasks, respectively, on the 28 threads of one ThunderX2 socket. Both timelines show the same duration. In the Loop timeline, light green is communication and turquoise is stencil compute. In the Task-2 timeline, the colors are the same and dark purple refers to checksum tasks. The Loop timeline shows a clear imbalance across threads in the communication phase, with certain threads consistently doing less work than others due to working on blocks at different refinement levels. The Task-2 timeline shows communication, stencil and checksum tasks concurrently executing as they become ready, leading to incidental locality improvement and little idle time. This incidental locality improvement happens more often with lower thread counts (4-8). Some consumer tasks execute faster due to executing back-to-back with their producer, e.g., communications of a block happening right after its stencil computation, or vice versa. In the absence of a locality-aware scheduler, this is less likely on larger thread counts and we observe a larger drop in task performance.

Looking across systems, the Task-2 version results in over 90% useful time on threads, i.e., communication/stencil/checksum, with a few threads achieving just over 80% utilization due to task creation time not being accounted as useful. The Loop version gets a lower utilization of between 40% and 80%. The threads that spend more than half of the time idle are those that repeatedly operate on blocks at the lower refinement levels.

Figure 4 shows the normalized execution time on ThunderX2 using Arm Compiler, on POWER9 using IBM XL, and on Skylake-SP using Intel compiler.

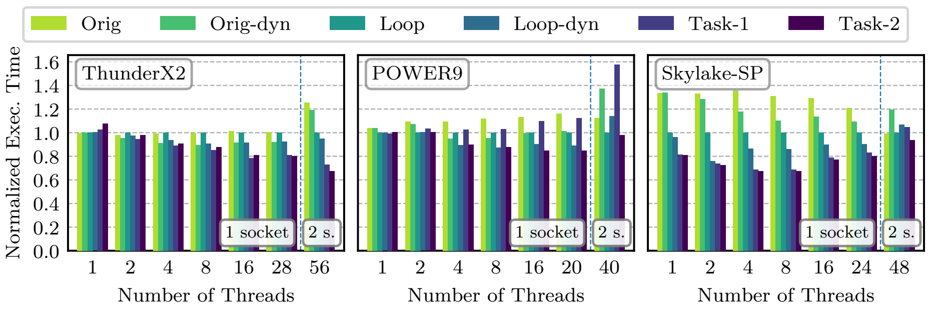

**Fig. 4.** Execution time with proprietary compilers: Arm Compiler on ThunderX2 (left), IBM XL on POWER9 (middle) and Intel Compiler on Skylake-SP (right)

The tasking versions achieve similar gains on ThunderX2 with the exception of dual socket which performs better. On POWER9, tasking gets smaller gains and Loop-dyn performs the same in certain thread counts. On Skylake-SP, the tasking advantage over the loop-parallel versions is even larger than with GNU.

Testing other application parameters to verify the sensitivity of this analysis showed some variations in the results, but they do not change the conclusions above. Going to arbitrarily small 8×8×8 blocks to stress task creation overhead, indeed shows smaller benefit of the task versions and they scale worse overall, especially across sockets where they perform significantly worse, but still work better than Loop within single socket cases. Going to 160 variables to isolate task creation overhead, and a checksum frequency of 10 for larger task-scheduling look-ahead, shows a bit better results for tasking but not significantly better than the ones using 40 variables or a checksum frequency of 5. Going to a checksum frequency of 10 instead of 5 also shows a bit better results for tasking and the benefit of Task-2 over Task-1 is also larger.

## 6 Conclusion

The benefits of tasking come mainly from a higher level view of parallelization by the programmer. Introducing asynchrony by parallelizing across program phases enables a higher utilization of threads thanks to removing global synchronizations, not having serial code between loops and, compared to static scheduling, avoiding load imbalance. Due to the lack of locality-aware scheduling in the tested runtimes (to the best of our knowledge), locality improvements by consumer tasks executing after producer tasks was incidental. Also, tasking suffers from worse NUCA/NUMA behavior because tasks operating on the same blocks may execute on different threads across sockets and chiplets. Our experiments suggest that locality/affinity semantic extensions for tasking in OpenMP have potential for significant performance improvement and scaling across NUMA domains if paired with balanced data allocation.

Parallelizing across program phases requires a mindset change if the programmer tends to parallelize loops or small sections after having parallelized at the MPI level. This strategy of focusing on small code portions when parallelizing with OpenMP limits scaling because sections between parallel regions remain serial. Tasking helps think in terms of larger code sections thanks to task dependences—a task can execute anytime during the task region as soon as its dependencies are satisfied.

A potentially-beneficial extension to the OpenMP standard for this taskification effort would have been the ability to specify dependences in taskloops. This way the Task-2 implementation could have been written in a easier and clearer way building on top of Task-1 code. This is an extension that is on-going work by the OpenMP committee and this paper shows a potential use case for it.

Lastly, we encountered several compiler issues with tasks that were reported to developers. Some compilers failed to compile certain constructs or generated incorrect results. These issues did not happen with worksharing-loop constructs, which shows the different maturity of both models.

**Acknowledgments** This work was in collaboration with Cray and funded in part by the DOE ECP PathForward program. It has been partially supported by the Spanish Government through Programa Severo Ochoa (contract SEV-2015-0493), by the Spanish Ministry of Economy and Competitiveness (contract TIN2015-65316-P), by the Generalitat de Catalunya (contracts 2017-SGR-1414 and 2017-SGR-1328), by the European Unions's Horizon 2020 Framework Programme under the Mont-Blanc project (grant agreement number 779877), and by the Arm-BSC Centre of Excellence initiative. I. Sánchez Barrera has been partially supported by the Spanish Ministry of Education, Culture and Sport under Formación del Profesorado Universitario fellowship number FPU15/03612. M. Casas has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramón y Cajal fellowship number RYC-2017-23269. M. Moretó has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramón y Cajal fellowship number RYC-2016-21104.

## References

1. Atkinson, P., McIntosh-Smith, S.: On the Performance of Parallel Tasking Runtimes for an Irregular Fast Multipole Method Application. In: International Workshop on OpenMP. pp. 92–106 (2017). https://doi.org/10.1007/978-3-319-65578-9_7
2. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Su, E., Unnikrishnan, P., Zhang, G.: A Proposal for Task Parallelism in OpenMP. In: International Workshop on OpenMP. pp. 1–12 (2007). https://doi.org/10.1007/978-3-540-69303-1_1
3. Berger, M.J., Colella, P.: Local adaptive mesh refinement for shock hydrodynamics. Journal of Computational Physics **82**, 64–84 (May 1989). https://doi.org/10.1016/0021-9991(89)90035-1
4. Berger, M.J., Oliger, J.: Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. Journal of Computational Physics **53**, 484–512 (Mar 1984). https://doi.org/10.1016/0021-9991(84)90073-1

5. Duran, A., Corbalán, J., Ayguadé, E.: Evaluation of OpenMP Task Scheduling Strategies. In: International Workshop on OpenMP. pp. 100–110 (2008). https://doi.org/10.1007/978-3-540-79561-2_9
6. Duran, A., Perez, J.M., Ayguadé, E., Badia, R.M., Labarta, J.: Extending the OpenMP Tasking Model to Allow Dependent Tasks. In: International Workshop on OpenMP. pp. 111–122 (2008). https://doi.org/10.1007/978-3-540-79561-2_10
7. ECP Proxy Apps Suite, https://proxyapps.exascaleproject.org/
8. Garcia-Gasulla, M., Mantovani, F., Josep-Fabrego, M., Eguzkitza, B., Houzeaux, G.: Runtime Mechanisms to Survive New HPC Architectures: A Use Case in Human Respiratory Simulations. The International Journal of High Performance Computing Applications (Apr 2019). https://doi.org/10.1177/1094342019842919
9. Gautier, T., Pérez, C., Richard, J.: On the Impact of OpenMP Task Granularity. In: International Workshop on OpenMP for Evolving Architectures. pp. 205–221 (Sep 2018). https://doi.org/10.1007/978-3-319-98521-3_14
10. Heroux, M.A., Doerfler, D.W., Crozier, P.S., Willenbring, J.M., Edwards, H.C., Williams, A., Rajan, M., Keiter, E.R., Thornquist, H.K., Numrich, R.W.: Improving Performance via Mini-applications. Tech. Rep. SAND2009-5574, Sandia National Laboratories (2009), http://www.mantevo.org/MantevoOverview.pdf
11. Mantevo Project, https://mantevo.org/
12. MiniAMR Adaptive Mesh Refinement (AMR) Mini-app, https://github.com/Mantevo/miniAMR
13. Rico, A., Ramirez, A., Valero, M.: Available Task-level Parallelism on the Cell BE. Scientific Programming **17**(1-2), 59–76 (2009). https://doi.org/10.3233/SPR-2009-0269
14. Sasidharan, A., Snir, M.: MiniAMR - A miniapp for Adaptive Mesh Refinement. Tech. rep., University of Illinois Urbana-Champaign (2016), http://hdl.handle.net/2142/91046
15. Teruel, X., Klemm, M., Li, K., Martorell, X., Olivier, S.L., Terboven, C.: A Proposal for Task-Generating Loops in OpenMP. In: International Workshop on OpenMP. pp. 1–14 (2013). https://doi.org/10.1007/978-3-642-40698-0_1
16. Vaughan, C.T., Barrett, R.F.: Enabling Tractable Exploration of the Performance of Adaptive Mesh Refinement. In: 2015 IEEE International Conference on Cluster Computing. pp. 746–752 (2015). https://doi.org/10.1109/CLUSTER.2015.129
17. Vidal, R., Casas, M., Moretó, M., Chasapis, D., Ferrer, R., Martorell, X., Ayguadé, E., Labarta, J., Valero, M.: Evaluating the Impact of OpenMP 4.0 Extensions on Relevant Parallel Workloads. In: International Workshop on OpenMP. pp. 60–72 (2015). https://doi.org/10.1007/978-3-319-24595-9_5
18. Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., Gautier, T.: Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. In: International Workshop on OpenMP. pp. 16–29 (2014). https://doi.org/10.1007/978-3-319-11454-5_2