Leveraging iterative applications to improve the scalability of task-based programming models on distributed systems

OMAR SHAABAN, JULIETTE FOURNIS D'ALBIAT, ISABEL PIEDRAHITA, VICENÇ BELTRAN, XAVIER MARTORELL, PAUL CARPENTER, EDUARD AYGUADÉ, and JESUS LABARTA, Barcelona Supercomputing Center, Spain

Distributed tasking models such as OmpSs-2@Cluster, StarPU-MPI, and PaRSEC express HPC applications as task graphs with explicit dependencies. The single task graph unifies the representation of parallelism across CPU cores, accelerators, and distributed-memory nodes, offering higher programmer productivity compared to traditional MPI + X. Most task-based models construct the task graph sequentially, which provides a clear and familiar programming model, simplifying code development, maintenance and porting. However, this design introduces a bottleneck in task creation and dependency management, limiting performance and scalability. As a result, unless the tasks are very coarse-grained, current distributed sequential tasking models cannot match the performance of MPI + X.

Many scientific applications, however, are iterative in nature, constructing the same directed acyclic task graph at each timestep. We exploit this structure to eliminate the sequential bottleneck and control message overhead in a sequentially-constructed distributed tasking model, while preserving its simplicity and productivity. Our approach builts on the recently proposed taskiter directive for OpenMP and OmpSs-2, allowing a single iteration to be expressed as a cyclic graph. The runtime partitions the cyclic graph across nodes, precomputes the MPI transfers, and then executes the loop body at low overhead. By integrating the MPI communications directly into the application's task graph, our approach naturally overlaps computation and communication, in some cases exposing dramatically more parallelism than fork–join MPI + OpenMP. We define the programming model and describe the full runtime implementation, and integrate our proposal into OmpSs-2@Cluster. We evaluate it using five benchmarks on up to 128 nodes of the MareNostrum 5 supercomputer. For applications with fork–join parallelism, our approach has performance similar to fork–join MPI + OpenMP, making it a viable productive alternative, unlike the existing OmpSs-2@Cluster model, which is up to 7.7 times slower than MPI + OpenMP. For a 2D Gauss–Seidel stencil computation, our approach enables 3D wavefront computation, giving performance up to 22 times faster than fork–join MPI + OpenMP and on-a-par with state-of-the-art TAMPI + OmpSs-2. All software, comprising the compiler, runtime and benchmarks, is released open source.[1]

CCS Concepts: • **Computing methodologies** → **Distributed programming languages**; **Distributed computing methodologies**; • **Software and its engineering** → **Distributed programming languages**.

Additional Key Words and Phrases: OpenMP, OmpSs-2@Cluster, MPI, Distributed Computing, Runtime Systems, Task-based Programming Models, Iterative Methods

[1]New Paper, Not an Extension of a Conference Paper

Authors' Contact Information: Omar Shaaban, omar.ibrahim@bsc.es; Juliette Fournis d'Albiat, juliette.fournis@bsc.es; Isabel Piedrahita, isabel.piedrahita@bsc.es; Vicenç Beltran, vicenc.beltran@bsc.es; Xavier Martorell, xavier.martorell@bsc.es; Paul Carpenter, paul.carpenter@bsc.es; Eduard Ayguadé, eduard.ayguade@bsc.es; Jesus Labarta, Barcelona Supercomputing Center, Barcelona, Spain, jesus.labarta@bsc.es.

Manuscript submitted to ACM

ACM Reference Format:
Omar Shaaban, Juliette Fournis d'Albiat, Isabel Piedrahita, Vicenç Beltran, Xavier Martorell, Paul Carpenter, Eduard Ayguadé, and Jesus Labarta. 2025. Leveraging iterative applications to improve the scalability of task-based programming models on distributed systems. *ACM Trans. Arch. Code Optim.* 1, 1 (May 2025), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Distributed-memory tasking approaches, such as OmpSs-2@Cluster [4], StarPU-MPI [9], PaRSEC [30], OMPC [61] and others [33], represent applications as tasks with dependencies. The runtime system builds a task graph and schedules tasks to execute concurrently across the available compute nodes. By managing data distribution and communication transparently, these approaches alleviate many of the complexities associated with MPI+X models [50, 51, 55], such as synchronization and potential deadlocks. Additionally, these models can be extended to support dynamic load balancing [5, 33] and both core- and node-level malleability [37]. While some frameworks, like PaRSEC's original Parameterized Task Graph (PTG) [24], rely on implicit task graphs, most adopt the Sequential Task Flow (STF) model, where the task graph is constructed dynamically as the execution progresses. STF provides clear and familiar semantics, simplifying development and facilitating the porting of existing codes.

The biggest issue with the distributed STF approach is limited scalability for medium- and fine-grained tasks. Depending on the benchmark and task granularity, OmpSs-2@Cluster, the distributed-memory variant of OmpSs-2 [12], scales to about 16 nodes [37]. In this approach, the task graph is constructed by the first rank; tasks are either executed locally or offloaded to other nodes for execution. OmpSs@cloudFPGA [25] and StarPU-MPI[2] avoid the offloading overhead by constructing an identical full task graph on every node. Each node retains only the tasks assigned to that node, together with the relevant communication. Since this requires every node to independently compute consistent mappings of task to node, this mapping must be static or deterministic, making it impossible to transparently balance the load. These approaches also ultimately suffer from the same sequential task and dependency management overhead.

Many scientific applications employ iterative methods or multi-step simulations, which execute an identical directed acyclic graph (DAG) of tasks at each timestep. A recent paper proposed the taskiter directive for OpenMP and OmpSs-2 [7], which specifies that a given loop consistently generates the same DAG of tasks and memory accesses in each iteration. It also asserts that the program remains correct if the loop body's non-task code executes only once. Using this information, the runtime executes the loop body once to generate tasks and dependencies for a single iteration. It then constructs a compact cyclic graph representation that describes the complete computation. As a result, the overhead of task creation, scheduling and dependency management are incurred only for the first iteration and amortized across all subsequent loop iterations. Tasks execute as if the loop were fully unrolled, without barriers between iterations.

Taskiter was proposed for SMPs, but it is an especially good fit for distributed tasking, for two main reasons. Firstly, as the number of nodes grows, the number of tasks typically grows in proportion, in order to occupy the computational resources, so the sequential bottleneck that motivates taskiter scales at least as fast as the number of nodes. Meanwhile, the wallclock time for the computation either stays roughly constant (for weak scaling, i.e., fixed computation per node) or it falls with the number of nodes (for strong scaling, i.e., fixed problem size). The result is that the growing sequential bottleneck quickly dominates the execution time. Secondly, knowledge of the full cyclic dependency graph ahead of time allows the runtime to precompute all MPI data transfers. With the usual approach,

[2]Note: StarPU supports two approaches: explicit communication inserted by the developer and implicit communication managed by the runtime system. In this context, we focus on the implicit communication mode. See Section 8 for more details.

these data transfers need to be computed dynamically, as the graph is built, resulting in a large number of control messages (see Section 2.2.2).

We extend the OmpSs-2@Cluster distributed tasking model to support a distributed form of taskiter and describe the full implementation. Our approach eliminates control messages to and from the master node, replacing them with peer-to-peer non-blocking MPI calls that are integrated into the application's task graph transparently. By integrating the MPI communications directly into the application's task graph, our approach naturally overlaps computation and communication, in some cases exposing dramatically more parallelism than fork–join MPI + OpenMP.

We implemented it in the Nanos6@Cluster runtime system and evaluated it using five benchmarks with problem sizes that scale up to 128 nodes of the MareNostrum 5 supercomputer. Our results improve on the existing OmpSs-2@Cluster approach, which, for 256 processes on 128 nodes, is slower than MPI + OpenMP, by between 7 and 34 times for four of the five benchmarks. In contrast, our maximum slowdown with respect to MPI + OpenMP is just 30%. For four of the five benchmarks, our performance either exceeds that of MPI + OpenMP or is within 8.9%. For the 2D heat equation stencil calculation, discussed in Section 3, which has the potential for 3D wavefront parallelism, we achieve 22 times higher performance than fork–join MPI + OpenMP, on-a-par with state-of-the-art asynchronous Task Aware MPI (TAMPI) + OmpSs-2. For the sake of fairness, none of these applications is unbalanced, so all performance improvements come from mitigating overheads and from the ability of task-based models to uncover more parallelism and overlap computation and communication. We expect even higher benefits from unbalanced applications. All software, including the runtime system and benchmarks is released open source [13].

In summary, the contributions of this paper are:

- We propose an extension for OmpSs-2@Cluster to improve its efficiency for iterative applications.
- We describe the runtime implementation and related optimizations.
- We implement the model in Nanos6@Cluster and give experimental results on up to 128 nodes of MareNostrum 5.
- We demonstrate that while the existing OmpSs-2@Cluster approach is only viable up to about 16 nodes, our approach is on-a-par with an asynchronous TAMPI + tasks hand-optimized implementation up to at least 128 nodes.

The rest of this paper is structured as follows. Section 2 is a summary of the relevant background. Section 3 is a motivational example of a 2D Gauss–Seidel stencil computation. Section 4 presents the programmer's model and Section 5 describes the runtime implementation in detail. Section 6 describes the evaluation methodology and Section 7 presents the experimental results. Finally, Section 8 discusses the related work and Section 9 concludes the paper.

2 BACKGROUND

2.1 OmpSs-2, Nanos6 and LLVM

OmpSs-2 [12] is the second version of the OmpSs [27] programming model, which serves as a research platform for developing and evaluating ideas that could be proposed for standardization in OpenMP [10, 42]. OmpSs emphasizes task-based parallelism and dataflow [46], targetting heterogeneous architectures by encapsulating native kernels within asynchronous tasks.

OmpSs-2 extends OmpSs to improve task nesting and enable fine-grained dependencies across nesting levels [8, 45]. This is achieved through the use of weak access types. The reference implementation of OmpSs-2 comprises the OmpSs-2 compiler [16], which is based on LLVM [3]/Clang [2], and the Nanos6 [11] runtime.

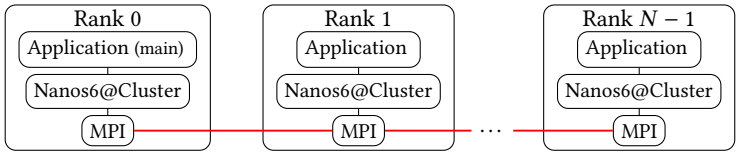
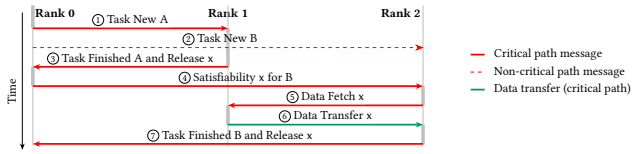


Fig. 1. OmpSs-2@Cluster architecture. The main function is a task on Rank 0. All other tasks may be offloaded to any other rank.

```
1 #pragma oss task depend(weakout:x) node(1)
2 { ··· } // Task A
3 #pragma oss task depend(weakin:x) node(2)
4 { ··· } // Task B
```

(a) Source code



(b) MPI messages

Fig. 2. MPI messages for OmpSs-2@Cluster. Offloading and executing two tasks requires one data transfer and six control messages.

2.2 OmpSs-2@Cluster

OmpSs-2@Cluster extends OmpSs-2 to enable asynchronous dataflow task execution across nodes through task offloading [4, 14]. The framework supports active malleability by interacting with the job scheduler to request or release compute nodes, with resource management and data distribution occuring transparently to the programmer [37]. Additionally, OmpSs-2@Cluster enables multi-node dynamic load balancing for MPI + OmpSs-2 programs [5], utilizing its task offloading capabilities to ensure a balanced workload across nodes.

As shown in Figure 1, each rank runs an instance of the Nanos6@Cluster runtime. The runtimes coordinate as peers, with all communication for control messages and data transfers done using two-sided MPI point-to-point communication. All runtimes mmap a common virtual memory region, so that data allocated on any node can be accessed, at the same address, by any other task on any node, so long as it is part of the task's access specification.

2.2.1 Tasks, offloading and scheduling. The main function runs as a task on Rank 0, and all other tasks are created as a subtask of their parent, which may be main or any other task. Initially, the subtask is created on the same rank as its parent. Once its dependencies are satisfied and it becomes ready for execution, scheduling occurs in two stages. First, the cluster scheduler on the parent's rank determines which rank should execute the task. If the task is assigned to a different rank, an MPI message is sent to the designated remote rank, which then creates a copy of the task. In the second scheduling stage, the host scheduler on the local or remote execution rank schedules the task to run on an available core.

Optimized OmpSs-2@Cluster programs (without distributed taskiter) typically employ two levels of nested tasks. At the top level, there is a single offloadable task per rank to distribute work across processes, while at the second level, a small number of non-offloadable tasks per core distribute work across the available cores. This approach helps mitigate the bottleneck caused by sequential task creation and offloading, but it introduces additional complexity for the programmer. Since the top-level tasks do not themselves perform computation, they can use weak accesses (as described in Section 2.1), allowing them to execute and create their subtasks before the data is available.

In order to provide a fair comparison with the best possible performance for original OmpSs-2@Cluster, the OmpSs-2@Cluster benchmarks, without distributed taskiter, in this paper use two levels of nested tasks (see Section 6). In contrast, the distributed taskiter benchmarks do not rely on task nesting.

```
1  #pragma omp taskiter [clause [...]] new-line
2        loop
```

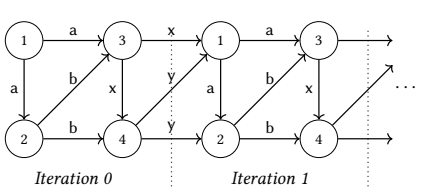
(a) Taskiter clause (OpenMP)

```
1  #pragma oss taskiter [clause [...]] new-line
2        loop
```

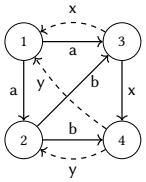
(b) Taskiter clause (OmpSs-2)

```
1  #pragma omp taskiter
2  for(int it=0; it<NUM_ITERATIONS; it++) {
3      #pragma omp task depend(in:x,y) depend(out:a)
4      { ··· } // Task 1
5      #pragma omp task depend(in:a,y) depend(out:b)
6      { ··· } // Task 2
7      #pragma omp task depend(in:a,b) depend(out:x)
8      { ··· } // Task 3
9      #pragma omp task depend(in:x,b) depend(out:y)
10     { ··· } // Task 4
11 }
```

(c) Source code



(d) Task dependence graph without taskiter



(e) DCTG with taskiter

Fig. 3. Example OpenMP program using taskiter. The taskiter annotation on line 1 of subfigure (a) enables the runtime to replace the normal unrolled task graph in subfigure (b) with the concise directed cyclic task graph (DCTG) illustrated in subfigure (c).

2.2.2 Control messages and global write ordering. Figure 2a shows an OmpSs-2@Cluster program that creates two tasks with weak accesses. The program begins execution on Rank 0, and, for concreteness, the node() clause is used to overrule the cluster scheduler by explictly assigning each task to a specific execution rank. Figure 2b illustrates the sequence of MPI messages that OmpSs-2@Cluster uses to offload these tasks and enforce dependencies. First, the tasks are offloaded with two consecutive Task New messages, sending Task A to Rank 1 and Task B to Rank 2. Once Rank 1 completes Task A, it sends a Task Finished message back to Rank 0, releasing the output value of x and identifying the location of the latest data version. Rank 0 then sends a Satisfiability message to Rank 2, granting global write permission and providing the current location of x. Since the data resides on Rank 1, Rank 2 sends a Data Fetch message to Rank 1, which responds with a point-to-point data transfer. Finally, when Task B completes, Rank 2 sends a Task Finished message back to Rank 0. In total, the runtimes exchange seven messages, with all but one (the offloading of Task B) on the critical path. Only a single MPI message carries the actual data transfer.

2.3 Taskiter

The *taskiter* construct was recently proposed [7] for OmpSs-2 and OpenMP. Figures 3a and 3b show the syntax for OpenMP and OmpSs-2, respectively. Any loop can be annotated with taskiter, provided that (a) each iteration generates the same top-level dependency graph of tasks and accesses, and (b) the program remains valid if the code inside the loop body but outside any task is executed just once. Condition (a) applies only to the top-level dependency graph: each iteration of a top-level task may create a distinct task graph of subtasks, without restriction.

An example program using taskiter is shown in Figure 3c, where the only modification to take advantage of taskiter is the pragma annotation on line 1. Figure 3d shows the regular task graph. When using taskiter, however, the runtime only executes one iteration of the taskiter's loop, creating the dependency graph for a single iteration. It then converts the graph into the directed cyclic task graph (DCTG) shown in Figure 3e.

In this figure, the non-cyclic edges, between tasks in the same iteration, are shown as solid lines and the cyclic edges, passing from one iteration to the next, are shown as dashed curves. By only creating tasks and computing dependencies for a single iteration, taskiter greatly reduces the sequential overhead. Moreover, since the DCTG is constant for all iterations, it is stored in a simple form, without locking or complex lock-free data structures. The tasks are still executed as in Figure 3d, without barriers between iterations.

The optional unroll(n) clause allows taskiter to handle loops where the task graph repeats every n iterations. A common example is a loop where each iteration reads from one array, writes to another, and then swaps the pointers to alternate the roles of the input and output arrays. While the original loop generates the same tasks in each iteration,

```
1  double x[10];
2  #pragma omp task depend(out:x)
3  {
4    MPI_Request request;
5    MPI_Irecv(&x, 10, MPI_DOUBLE, other_rank, tag, MPI_COMM_WORLD, &request);
6    TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
7  }
8  #pragma omp task depend(in:x)
9  { ··· }
```

Fig. 4. Example OpenMP program using Task-aware MPI's (TAMPI's) non-blocking communication. The call to TAMPI_Iwait delays the release of the current task's dependencies until the MPI call on line 5 completes.

pointer swapping causes the data accesses to differ in even and odd iterations. With the unroll clause, the loop executes n times instead of once, generating an unrolled task graph for the n iterations. For instance, unrolling the loop with pointer swapping by a factor of two ensures that each iteration now generates the same graph of tasks and accesses, thus meeting the requirements for taskiter. There is no benefit to unrolling beyond the minimum necessary for correctness. Additional unrolling would increase the initial overhead without affecting the steady-state throughput.

If the compiler cannot determine the total number of loop iterations at the start of the loop, then it inserts a special task known as a control task. The control task depends on every subtask in the current iteration as well as the control task from the previous iteration. The body of the control task checks the loop's condition and it cancels the taskiter when the condition is false. When the taskiter has the unroll clause, these control tasks are strided by the unrolling factor, providing a means to overlap tasks from different iterations.

2.4 Task-aware MPI (TAMPI)

Hybrid MPI+X applications are typically structured as alternating fork–join computation and sequential communication phases. This incurs additional synchronization, which, as we confirm in our results, can hinder inter- and intra-node parallelism. We evaluate our approach primarily in comparison with fork–join MPI+OpenMP, but, since OmpSs-2@Cluster is naturally asynchronous, we also compare with state-of-the-art asynchronous TAMPI.

TAMPI [51] is a library that integrates blocking and non-blocking MPI primitives with task-based programming models. It introduces a new level of MPI threading support, known as MPI_TASK_MULTIPLE. An application that requests this threading level can use blocking MPI primitives inside tasks without the risk of data races or deadlock. The existing MPI_THREAD_MULTIPLE threading level causes blocking MPI primitives to block the underlying thread running the task. Even if the MPI implementation avoids busy waiting and idles the hardware thread, the task-based runtime cannot discover that the hardware thread is available. TAMPI uses the PMPI interface to intercept MPI calls, and it releases any blocking thread to the runtime system, so that it can execute other tasks, thereby preventing deadlock.

TAMPI also simplifies and optimizes the use of non-blocking MPI primitives, by making their completion visible to the dependency system. This is done using the new TAMPI_Iwait and TAMPI_Iwaitall calls, as illustrated in Figure 4. The task on line 2 posts a non-blocking MPI_Irecv on line 5. It then calls TAMPI_Iwait, on line 6, which informs TAMPI that the given MPI request is associated with a dependency to the subsequent task (it comprises the output dependency on x). TAMPI uses the Nanos6 external events API [50] to postpone the release of the current task's dependencies. The call to TAMPI_Iwait is non-blocking, so the task continues, finishing immediately and freeing its data structures and stack. Later, when the MPI request completes, TAMPI will use the external events API to release the task's dependencies, an action that, unlike TAMPI's blocking MPI receive, does not involve unblocking and rescheduling the first task. At this point the task on line 8 becomes ready for execution.

3 MOTIVATION

This section motivates our work through three variants of a Gauss–Seidel 2D heat equation, shown in Figure 5: fork–join MPI + OpenMP, asynchronous TAMPI + OpenMP/OmpSs-2, and OmpSs-2@Cluster with distributed taskiter. The benchmark is an in-place 2D stencil calculation that updates each element based on the values of the elements above and to the left from the current timestep and to the right and below from the previous timestep. The matrix is of size NBY × NBX blocks, each of size BSY × BSX elements. It is distributed among the processes cyclically by rows.

Figure 5a shows an implementation using fork–join parallelism with MPI and OpenMP. The local part of the stencil calculation has NBY_LOCAL rows, including the single-row halos at the top and bottom. The computation is done using tasks with dependencies (lines 29 to 38 in Figure 5a) in order to update all elements in a timestep using 2D wavefront parallelism. The parallelism rises from zero at the beginning of the timestep up to a maximum of the number of blocks along the shortest dimension. It then drops back down to zero at the end of the timestep, due to the taskwait on line 39.

Figure 5b shows an asynchronous TAMPI + OpenMP implementation. This version eliminates the taskwait between timesteps, and it has higher performance due to 3D wavefront parallelism, which allows concurrent execution of tasks from different timesteps. The parallelism rises from zero at the beginning of the first timestep, and it stays at the maximum value until close to the end of the last timestep. The cost is extra complexity to encapsulate the sends and receives into tasks and use TAMPI's non-blocking API (see Section 2.4).

Figure 5c shows the OmpSs-2@Cluster implementation using taskiter. Whereas most of the code in the MPI + OpenMP and TAMPI + OpenMP versions concern orchestration and micro-management of data distribution and communication, only two declarative pieces of information have been introduced to the OmpSs-2@Cluster version. First, the call to nanos6_set_affinity on line 5 describes the data affinity. This call does not move data, but it hints to the runtime that the non-readonly rows of the matrix ought to be distributed cyclically across the ranks. Second, the accesses to the blocks above and below the current block have been made more precise, since each task reads only a single row of these blocks, rather than the whole block. Without this change, the oversized data accesses would mislead the runtime and force data transfers between neighboring ranks to exchange whole blocks, each of size BSY×BSX elements. The explicit MPI calls in the MPI + OpenMP and TAMPI + OpenMP/OmpSs-2 have already been optimized to exchange single rows of size BSX elements. The modification to the OmpSs-2@Cluster version is simple, due to the fragmented regions dependency system, which correctly computes task dependencies even when tasks have data accesses that overlap in arbitrary ways. The overhead of the fragmented regions dependency system only affects the initial overhead of a distributed taskiter, since the steady-state execution is done on a simplified cyclic graph of tasks (see Section 5.4.2). Nevertheless, the program could easily be rewritten using regular discrete dependencies.

Overall, the number of lines of code in the OmpSs-2@Cluster version with distributed taskiter is about about one third that of the TAMPI + OpenMP version, and almost all of the code relates to the actual Gauss–Seidel computation.

4 PROGRAMMER'S MODEL

The programmer's model for distributed taskiter is the same as described in Section 2.3, except for the rules related to the definition of accesses:

Full definition of accesses: OmpSs-2@Cluster requires all tasks to have a full specification of their accesses [4], so that the runtime can program any necessary data transfers. This requirement is inherited for taskiters, and it differs from the situation on SMP, where the taskiter only needs to be given accesses when necessary to enforce ordering with

```
double matrix[NBY_LOCAL][NBX][BSY][BSX];
int main(int argc, char **argv)
{
 int provided;
 MPI_Init_thread(argc, argv, MPI_THREAD_MULTIPLE, &provided);
 assert(provided == MPI_THREAD_MULTIPLE);
 ...
 for (int it=0; it < NUM_ITERATIONS; it++) {
  MPI_Request request[NBX*3];
  int count = 0;
  if (rank != 0) { // Send first compute row
   for (x=1; x < NBX-1; x++) {
    MPI_Isend(&matrix[1][x][0][BSY-1], BSX, MPI_DOUBLE, rank-1,
          bx+it*NBX, MPI_COMM_WORLD, &request[count++]);
   }
   // Receive upper border
   for (x=1; x < NBX-1; x++) {
    MPI_Irecv(&matrix[0][x][0][BSY-1], BSX, MPI_DOUBLE, rank-1,
          bx+it*NBX, MPI_COMM_WORLD, &request[count++]);
   }
  }
  if (rank != rank_size - 1) { // Receive lower border
   for (x=1; x < NBX-1; x++) {
    MPI_Irecv(&matrix[NBY_LOCAL-1][x][0], BSX, MPI_DOUBLE, rank+1,
          bx+it*NBX, MPI_COMM_WORLD, &request[count++]);
   }
  }
  MPI_Waitall(count, request, MPI_STATUSES_IGNORE);
  for (int y=1; y<NBY_LOCAL-1; y++) {
   for(int x=1; x<NBX-1; x++) {
    #pragma omp task depend(inout:matrix[y][x]) \
    depend(in:matrix[y-1][x]) depend(in:matrix[y][x-1]) \
    depend(in:matrix[y][x+1]) depend(in:matrix[y+1][x])
    {
     GaussSeidelBlock(matrix, x, y);
    }
   }
  }
  #pragma omp taskwait
  if (rank != rank_size -1) {
   // Send last compute row
   count = 0;
   for (x=1; x < NBX-1; x++) {
    MPI_Isend(&matrix[NBY_LOCAL-2][x][0], BSX, MPI_DOUBLE, rank-1,
          bx+it*NBX, MPI_COMM_WORLD, &request[count++]);
   }
   MPI_Waitall(count, request, MPI_STATUSES_IGNORE);
  }
 }
}
```

(a) Fork–join MPI + OpenMP tasks

```
double matrix[NBY][NBX][BSY][BSX];
int main(int argc, char **argv)
{
 ...
 nanos6_set_affinity(&matrix[1], (NBY-2) * NBX * BSY, BSX,
        nanos6_equpart_distribution, 0, NULL);
 #pragma oss taskiter depend(weakinout:matrix)
 for (int it=0; it < NUM_ITERATIONS; it++) {
  for (int y=1; y<NBY-1; y++) {
   for (int x=1; x<NBX-1; x++) {
    #pragma oss task depend(inout:matrix[y][x]) \
    depend(in:matrix[y-1][x][BSY-1]) depend(in:matrix[y][x-1]) \
    depend(in:matrix[y][x+1]) depend(in:matrix[y+1][x][0])
    {
     GaussSeidelBlock(matrix, x, y);
    }
   }
  }
 }
}
```

(c) OmpSs-2@Cluster taskiter

```
double matrix[NBY_LOCAL][NBX][BSY][BSX];
int main(int argc, char **argv)
{
 int provided;
 MPI_Init_thread(argc, argv, MPI_TASK_MULTIPLE, &provided);
 assert(provided == MPI_TASK_MULTIPLE);
 ...
 for (int it=0; it < NUM_ITERATIONS; it++) {
  if (rank != 0) { // Send first compute row
   for (x=1; x < NBX-1; x++) {
    #pragma omp task depend(in:matrix[1][x])
    {
     MPI_Request request;
     MPI_Isend(&matrix[1][x][0][BSY-1], BSX, MPI_DOUBLE, rank-1,
           bx+it*NBX, MPI_COMM_WORLD, &request);
     TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
    }
   }
   // Receive upper border
   for (x=1; x < NBX-1; x++) {
    #pragma omp task depend(out:matrix[0][x])
    {
     MPI_Request request;
     MPI_Irecv(&matrix[0][x][0][BSY-1], BSX, MPI_DOUBLE, rank-1,
           bx+it*NBX, MPI_COMM_WORLD, &request);
     TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
    }
   }
  }
  if (rank != rank_size - 1) { // Receive lower border
   for (x=1; x < NBX-1; x++) {
    #pragma omp task depend(out:matrix[0][x])
    {
     MPI_Request request;
     MPI_Irecv(&matrix[NBY_LOCAL-1][x][0], BSX, MPI_DOUBLE, rank+1,
           bx+it*NBX, MPI_COMM_WORLD, &request);
     TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
    }
   }
  }
  for (int y=1; y<NBY_LOCAL-1; y++) {
   for(int x=1; x<NBX-1; x++) {
    #pragma omp task depend(inout:matrix[y][x]) \
    depend(in:matrix[y-1][x]) depend(in:matrix[y][x-1]) \
    depend(in:matrix[y][x+1]) depend(in:matrix[y+1][x])
    {
      GaussSeidelBlock(matrix, x, y);
    }
   }
  }
  if (rank != rank_size -1) { // Send last compute row
   for (x=1; x < NBX-1; x++) {
    #pragma omp task depend(in:matrix[0][x])
    {
     MPI_Request request;
     MPI_Isend(&matrix[NBY_LOCAL-2][x][0], BSX, MPI_DOUBLE, rank-1,
           bx+it*NBX, MPI_COMM_WORLD, &request);
     TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
    }
   }
  }
 }
}
```

(b) Asynchronous TAMPI + OpenMP/OmpSs-2

Fig. 5. Gauss–Seidel 2D heat equation using three different programming models. The OmpSs-2@Cluster version achieves similar performance to the TAMPI + OpenMP/OmpSs-2 version, with about one third the number of lines of code and much lower complexity.

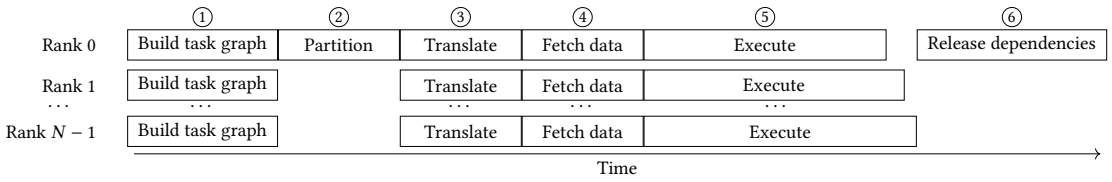


Fig. 6. Illustration of the execution process for a distributed taskiter. This figure gives an overview of the sequence of steps and where they are executed, but it does not quantify the relative durations of the steps, which are not to scale.

```
#pragma oss taskiter depend(weakinout:x,y,a,b)
for(int it=0; it<NUM_ITERATIONS; it++) {
  #pragma oss task depend(in:x,y) depend(out:a) node(0)
  { ··· } // Task 1
  #pragma oss task depend(in:a,y) depend(out:b) node(1)
  { ··· } // Task 2
  #pragma oss task depend(in:a,b) depend(out:x) node(0)
  { ··· } // Task 3
  #pragma oss task depend(in:x,b) depend(out:y) node(1)
  { ··· } // Task 4
}
```

Fig. 7. Distributed taskiter version of program of Figure 3c with the mapping from task to rank indicated using the "node" clause.

its sibling tasks. Any data that is only required by subtasks should be specified as a weak access (defined in Section 2.1). Any data that is required by the loop body or loop condition needs to be specified as a strong (i.e., non-weak) access.

Precise definition of accesses: The task accesses give a unified specification of the data accessed by the task, both for task ordering and to program data transfers. These accesses should precisely define the data that is needed by the task, in order to avoid unnecessary data transfers (an example was given in Section 3).

5 IMPLEMENTATION

The execution of a distributed taskiter is illustrated in Figure 6, which shows a timeline, from left to right, of the steps, ①, ②, ⋯, ⑥, executed by each rank. This figure is intended to give an overview of the process, rather than quantifying the relative durations of these steps, which are not to scale.

5.1 Compilation

The compiler encapsulates the loop body as a task and inserts Nanos6 API calls to create and submit the task, in a similar way to any other task defined in the program. The taskiter task also passes the loop bounds and a flag to identify the task as a taskiter. This is exactly the same as the compilation process for taskiter on SMP [7]. Apart from moving the loop body into a task, the compiler does not apply any transformations to the code.

5.2 Execute taskiter parent task to build full task graph

The taskiter becomes ready under the same conditions as any other task, i.e. once all of its strong accesses, if any, are satisfied. At that point, the original node (typically Rank 0) creates and offloads a parentless copy of the taskiter to the other ranks. As illustrated in Step ① of Figure 6, all ranks execute the taskiter task, so that each rank constructs an identical local copy of the full task dependency graph for a single iteration. If the taskiter has the *unroll* clause, this "single iteration" may correspond to multiple iterations of the original loop.

5.3 Partition graph among nodes

Once Rank 0 has finished executing the taskiter task, and has created all the subtasks, it performs Step ② of Figure 6, which partitions the dependency graph for execution by the processes. Our approach can leverage any partitioning algorithm, and it does not require a fixed or deterministic method, unlike StarPU-MPI [9] and OmpSs@cloudFPGA [25] (see Section 8). The current prototype uses a static partition controlled by the node clause.

Figure 7 updates the example program of Figure 3c to use OmpSs-2@Cluster with distributed taskiter, and it adds the node clause on each task to indicate the partition that will be used as an example in the rest of this section. Tasks 1 and 3 are executed on Rank 0 and Tasks 2 and 4 are executed on Rank 1, enabling wavefront parallelism across two nodes.

5.4 Translate to create local graph

Step ③ of Figure 6 translates the full dependency graph into the local directed cyclic task graph (DCTG) for execution by the current process, and it pre-computes the MPI data transfers involving the current process. This step is done concurrently by all ranks, as illustrated in Figure 6. There are two sub-steps: (1) insert communication tasks and (2) create the local DCTG.

5.4.1 Insert communication tasks. Communication among ranks is done by dedicated tasks, in order to exploit the existing task graph to orchestrate the ordering and overlap of communication and computation. A send task has an *in* access on the data being sent, since the MPI send operation only requires reading the latest version of the data. Conversely, a receive task has an *out* access, since the MPI receive operation updates its buffer with the new version of the data. The communication tasks are implemented to avoid deadlock, as detailed in Section 5.6.

The runtime algorithm for adding send and receive tasks is shown in Figure 8. It starts from the top map, an existing data structure that maps each region to the first task that accesses it.[3] The algorithm has a complexity of $O(R)$, where R is the total number of data accesses, computed as the sum over tasks of the number of regions accessed by the task. This value corresponds to the total number of iterations of the while loop on line 8 of Figure 8. When using the regions dependency system, an additional pass with the same $O(R)$ complexity is required to fully fragment the top map to match the finest access granularity.

Figure 9a shows the output of the algorithm of Figure 8 for Rank 0 of the partitioned program in Figure 7. Tasks created in Step ① of Figure 6 that will not be executed locally on Rank 0, i.e., Task 2 and Task 4 are grayed out. We assume that the virtual addresses of the variables are in the order a, b, x, y. The loop on line 3 processes each region in the top map in virtual address order, in this case starting with a. Next, the while loop on line 8 considers each task access that contains a, starting with the *out* access of Task 1. This is the first write to a ($lastWriter$ is none on line 20) so the empty set of $initialReaders$ is captured on line 21: since the first access is a write, it will be not necessary to perform data transfers for cyclic edges into Task 1. The next access to the same region is the *in* access of Task 2. This task reads the data, but it is not yet present on Rank 1 ($access.rank \notin readerRanks$ on line 9), so it requires a data transfer from Rank 0, the $lastWriter$, to Rank 1, which executes the task. Since the current rank is Rank 0, a send task is created on line 13. Following the same procedure, Rank 1, creates the matching receive task on line 15. This completes the accesses to a, so the loop on line 3 continues for b, which follows a similar process, except that the current rank, Rank 0, creates a receive task. The process for x is slightly different, because the first access to x is the *in* access of Task 1, which reads the value from the as-yet-unknown last writer in the previous iteration. No send–receive pair is created (lines 11 to 15

[3]The top map is always present using the fragmented regions dependency system, as a way to link between a task's predecessors and its subtasks, and for the discrete dependency system, it is inherited from the usual SMP taskiter support.

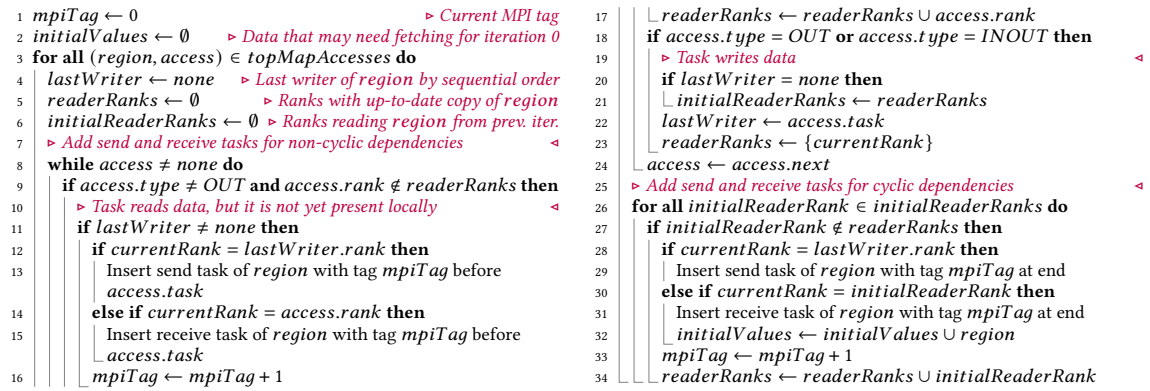


Fig. 8. Runtime algorithm to insert communication using asychronous send and receive tasks. This deterministic algorithm runs on all ranks on the same full task graph, so sends and receives on different ranks will always match.

are skipped), but Rank 0 is added to *readerRanks* on line 17. Later, once all accesses have been considered, the loop on line 26 will check the *initialReader*, Rank 0. Since *lastWriter* is also Rank 0, no send-receive pair is needed. Finally, considering y, the first access is the *in* of Task 0, and the loop on line 26 will also check the *initialReader*, Rank 0. But this time, since the *lastWriter* is Task 4 on Rank 1, there is no copy of this data on Rank 0, so a receive task for the cyclic read-after-write is created on line 31. This also means that iteration 0 on the current rank will require a valid copy of the data before the taskiter. This is recorded by adding the region to *initialValues* on line 32.

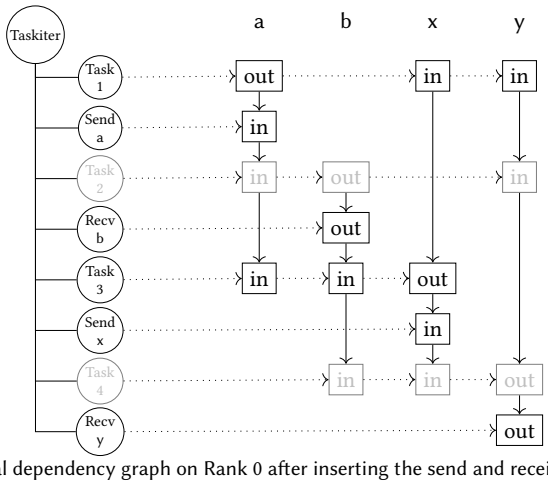
The iterations of send and receive tasks are serialized, irrespective of the algorithm used to execute the tasks.

Each receive is serialized due to the write-after-write dependency on its *out* access. Each send is serialized due to the write-after-read dependency from the send task (which has an *in* access) to the *lastWriter* (defined when the send task is created on line 13) in the next iteration, which has an *out* or *inout* access. MPI sends and receives of the same data in different iterations are therefore posted one at a time and in order. The MPI tags for corresponding sends and receives always match, since the sending and receiving ranks follow the same deterministic algorithm on the same task graph. The MPI tag is given by *mpiTag*, which is initialized to zero on line 1 and incremented on lines 16 and 33.

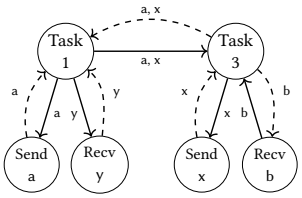
5.4.2 Create the local DCTG. The local dependency graph, which is a sequential graph on task accesses, is converted into the local directed cyclic task graph (DCTG). The result is a standard directed acyclic graph on tasks, as illustrated, for the example program, in Figure 9b.

5.5 Fetch input data

As soon as Step ③ of Figure 6 has finished on the current node, Step ④ fetches all of the input data required by the taskiter. The data regions that need fetching are identified by the value of *initialValues*, computed by the algorithm in Figure 8. Since the parent taskiter must have accesses covering all the accesses of its subtasks, the locations of the latest versions of these data regions are already known via the runtime's normal mechanism. Similarly, the runtime uses its normal mechanism to check whether data transfers are needed, merge contiguous data transfers and perform non-blocking MPI communication.



(a) Local dependency graph on Rank 0 after inserting the send and receive tasks. Tasks that are not executed on Rank 0 are disabled and colored in gray.



(b) Local DCTG on Rank 0.

Fig. 9. Regular dependency graph for a single iteration and directed cyclic task graph for Rank 0 of the example program of Figure 7.

5.6 Execute loop iterations

Once the data transfers in Step ④ of Figure 6, if any, have completed on the current node, Step ⑤ proceeds to execute all iterations of the body of the taskiter. It is not normally necessary to have a global barrier between Steps ④ and ⑤, but we add one in our experiments, in order to cleanly separate the startup overhead and the time per iteration. This extra barrier has little effect on the total execution time.

Communication tasks are ordinary tasks, except that the task body is implemented inside the runtime system rather than the user code. The body of the communication task simply posts the appropriate non-blocking MPI send or receive. The runtime defers the release of the dependencies to the successor tasks, which would otherwise happen immediately, until the MPI request completes. Completion of MPI requests is periodically tested by the same dedicated thread that is used for OmpSs-2@Cluster message completion [4].

5.6.1 Non-constant number of iterations. If the number of iterations is not known before the execution of the loop, then a control task is inserted, in a similar way to that described in Section 2.3.[4] The control task on Rank 0 inherits all the strong accesses of the taskiter, ignoring the weak accesses. The control task on Rank 0 also has a dependency on the previous iteration of the control task on the same rank. The control tasks on all the other ranks only have a dependency on the control task on Rank 0, which is used to copy the value of the condition to all other ranks. If the condition is false, then the control task on each rank cancels the rest of the taskiter. If the taskiter has the unroll clause, then the control task on Rank 0 is strided, in order to support overlapping of tasks from different iterations.

5.6.2 Scheduling policy. The runtime currently supports two scheduling policies. The **Iteration Priority** policy assigns higher priority to tasks from earlier iterations, ensuring uniform progress throughout the loop and maintaining a high level of parallelism until completion. In contrast, the **Immediate Successor** policy prioritizes tasks that consume data written by recently completed tasks, which typically improve data locality and cache utilization. However, if the DCTG

[4]Our prototype implementation does not yet support non-constant iteration counts.

Benchmark	Description	Parallelism	Communication	Problem size	Time per iter. on smallest #nodes
multi-matvec	Repeated dgemv	Independent tasks inside iteration	None	131072×131072	0.08 s (2 nodes)
multi-matmul	Repeated dgemm	Independent tasks inside iteration	None	65536×65536	82.93 s (4 nodes)
jacobi	Jacobi iteration [4]	Independent tasks inside iteration	All-to-all	131072×131072	0.04 s (4 nodes)
heat-gauss	2D stencil with Gauss–Seidel updates	2D/3D wavefront	Nearest-neighbor	131072×131072	0.49 s (2 nodes)
heat-jacobi	2D stencil with Jacobi updates	Independent tasks inside iteration	Nearest-neighbor	131072×131072	0.11 s (4 nodes)

Table 1. Evaluation benchmarks

is disconnected, some parts of the computation may finish early, leading to reduced parallelism as the remaining tasks are executed with limited concurrency towards the end of the computation.

5.7 Release accesses

Once all tasks executed on a remote (non-Rank 0) rank have completed all iterations, a Task Finished message is sent to Rank 0 as usual. After Rank 0 finishes its own iterations and has received notifications from all other nodes, it proceeds to Step ⑥ of Figure 6, which completes the taskiter and releases its accesses. At this point, Rank 0 updates the data locations in the dependency system to reflect the rank that executed the last writer. If no task performed a write operation because the data is only read, then the original location remains unchanged from before the taskiter.

6 EVALUATION METHODOLOGY

6.1 Hardware and software platform

The experiments in this paper were performed on up to 128 nodes of the general-purpose block of the MareNostrum 5 supercomputer [15]. MareNostrum 5 comprises 6408 compute nodes, each with two 56-core Intel Sapphire rapids sockets. We use the normal memory partition, which comprises 6192 nodes, each with 256 GB physical memory (2 GB per core). The interconnect is 100 Gb/s NVIDIA InfiniBand with a fat tree. GCC 13.2.0 was used to compile all benchmarks and the modified Nanos6@Cluster runtime. All benchmark kernels were in separate source files, identical for all programming models and compiled with the same compiler flags. The runtime uses Intel MPI 2023.2, and the benchmarks use the BLAS functions provided by Intel MKL 2023.2. All experiments use one MPI process per socket; i.e., two MPI processes per node. Each data point shows the average and standard deviation across five runs in different batch jobs, which we confirmed to have different node allocations. The same batch job was used to test all programming models using the same node allocation, to ensure fairness.

6.2 Benchmarks

The benchmarks are listed in Table 1. All are executed with one process per socket (i.e. 2 processes per node), following previous work [4], which found that this configuration improves NUMA locality, resulting in more stable and efficient performance. Strong scaling results are reported for 2 to 128 nodes, utilizing up to 14,336 cores across 256 MPI processes (with HyperThreading disabled). Each benchmark has a tunable block size that determines task granularity, and results are given for the block size that achieves the highest performance per iteration.

The last column of Table 1 gives the time per iteration for the best implementation of the benchmark on the smallest number of nodes with sufficient total memory to run the benchmark: 2 nodes for multi-matvec and heat-gauss, and 4 nodes for the others. With the exception of multi-matmul, for which the execution time is 82.9 s per iteration, all other benchmarks have a time per iteration, for the smallest configuration, of less than 0.5 s. This presents a challenging test for strong scalability of any distributed tasking system.

multi-matvec is a sequence of identical dense double-precision matrix–vector multiplications, with the matrix distributed by rows and without dependencies between iterations.[5]. It has fine-grained tasks with complexity $O(n^2)$ and no inter-node data transfers. multi-matmul is a sequence of dense double-precision matrix–matrix multiplications, with larger $O(n^3)$ tasks and also no inter-node data transfers. jacobi is an iterative double-precision Jacobi solver for dense strictly diagonally dominant systems. It is equivalent to repeatedly pre-multiplying a vector by a dense square matrix. It has the same $O(n^2)$ complexity as multi-matvec, but an all-to-all communication pattern, making it a particularly good fit for fork–join parallelism. heat-gauss is the Gauss–Seidel variant of the 2D heat equation stencil computation discussed in Section 3. It exhibits 2D or 3D wavefront parallelism and has the potential to overlap tasks from multiple iterations, making it a good fit for asynchronous task parallelism. heat-jacobi is the same 2D heat equation with Jacobi updates. This version has two working arrays and embarrassingly parallel computations inside each timestep to update the array. It is well suited to fork–join parallelism.

7 RESULTS

7.1 Strong scaling

Figure 10 shows the overall results for strong scaling. The five rows of charts correspond to the five benchmarks described in Section 6.2. In all plots the x-axis is the number of nodes, always with two MPI processes per node; i.e., one process per socket. The left-hand column of charts gives the initial overhead (seconds on the y-axis), which is independent of the number of iterations, and the right-hand column of charts gives the performance per iteration for the body of the loop (TFLOPS/s on the y-axis), excluding the initial overhead. This approach provides an overview of the overall behaviour, in a way that is independent of an arbitrary target number of iterations. To see the effect of the number of iterations on overall performance including overhead, as a function of the number of iterations, see Section 7.2.

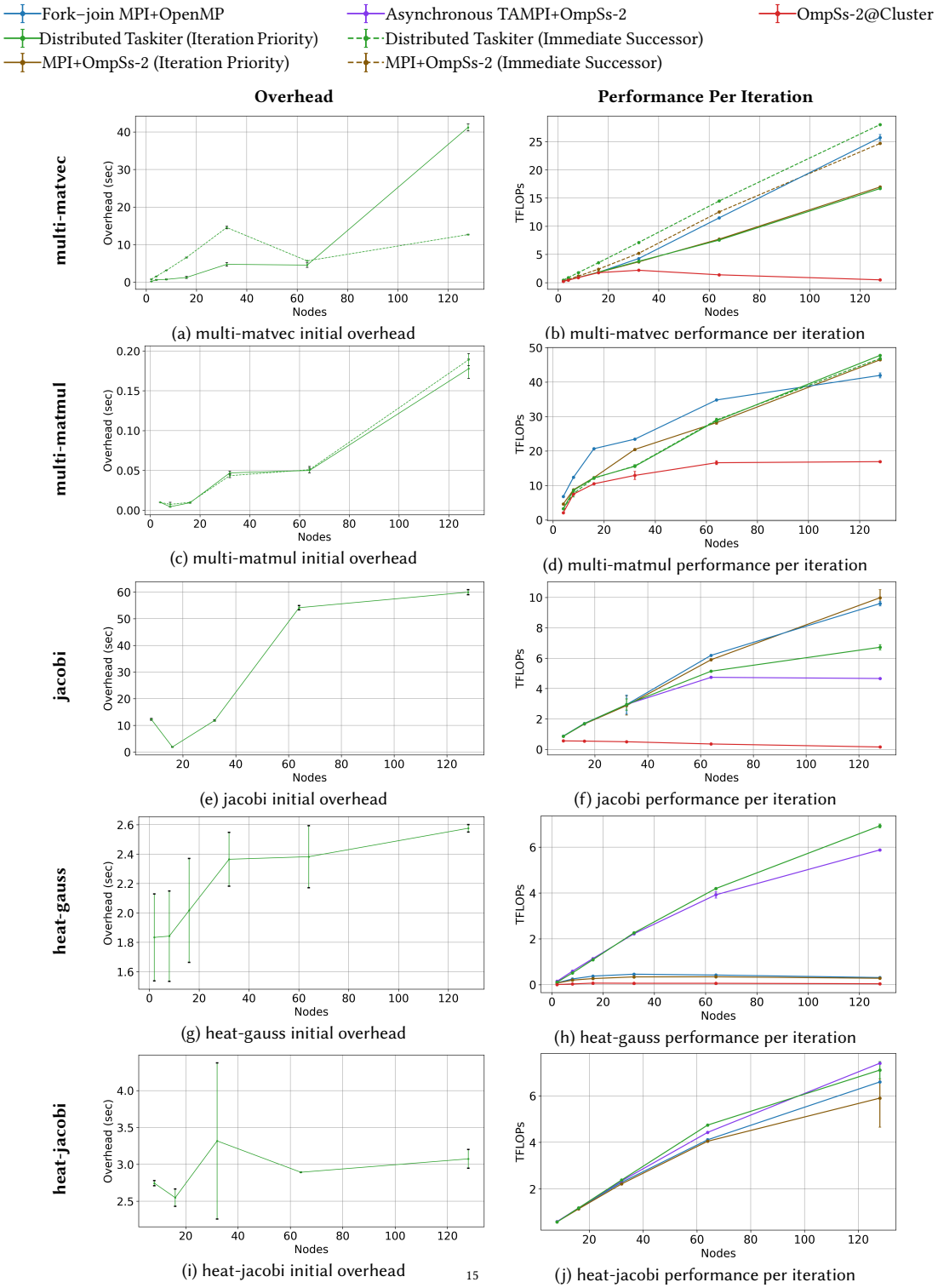
The colors in Figure 10 distinguish the five programming models: blue for fork–join MPI + OpenMP, purple for TAMPI + OmpSs-2, brown for fork–join MPI + OmpSs-2, red for the original OmpSs-2@Cluster, and green for distributed taskiter. For distributed taskiter, solid lines indicate the Iteration Priority policy and dashed lines indicate the Immediate Successor policy (see Section 5.6.2). All points use the block size that gives the best performance per iteration for the body of the loop, for that benchmark and implementation. All data points include error bars, showing the standard deviation across five runs on different allocated nodes.

Figure 10a shows the overhead for multi-matvec, which takes a maximum value of 40.0 seconds on 128 nodes. This overhead corresponds to Steps ① through ④ and Step ⑥ as described in Section 5. As detailed in Section 5.4.1 and further supported by empirical analysis in Section 7.3, the overhead scales approximately linearly in the total number of accesses. For this benchmark, the optimal block size results in a small number of tasks per core. Since each task has a fixed number of accesses, the total number of accesses should, in theory, scale linearly with the number of nodes. In practice, experimental noise and other factors prevent the optimal block size from perfectly adhering to the expected behavior, leading to the variability in the results presented in the chart.

Figure 10b shows the throughput per iteration for multi-matvec. The fork-join MPI + OpenMP and Immediate Successor versions of distributed taskiter and TAMPI + OmpSs-2 achieve comparable performance and scalability.[6] On 128 nodes, the distributed taskiter version is about 8.9% faster than the other two versions, due to the lower task management overhead. The scheduling policy of OpenMP is implementation defined, but we verified through Paraver traces that for this benchmark, OpenMP followed a similar schedule to Immediate Successor. This benchmark has a

[5]Prior work [4] referred to these same benchmarks as matvec and matmul

[6]Since there is no MPI communication in this benchmark, TAMPI + OmpSs-2 is equivalent to regular MPI + OmpSs-2.



(a) multi-matvec initial overhead

(b) multi-matvec performance per iteration

(c) multi-matmul initial overhead

(d) multi-matmul performance per iteration

(e) jacobi initial overhead

(f) jacobi performance per iteration

(g) heat-gauss initial overhead

(h) heat-gauss performance per iteration

(i) heat-jacobi initial overhead

(j) heat-jacobi performance per iteration

Fig. 10. Strong scaling results with two processes per node (one per socket). The five rows correspond to the five benchmarks. The left-hand column gives the initial overhead and the right-hand column gives the performance per iteration, excluding the initial overhead. All results use a block size for the best performance per iteration, this, in turn, affect the number of accesses, hence, we see the overhead of the distributed taskiter varies with maximum of 60 seconds (jacobi on 128 nodes). Distributed taskiter has performance per iteration that matches or exceeds fork–join MPI + OpenMP and is on-a-par with asynchronous TAMPI + OmpSs-2. The performance per iteration far exceeds that of the original OmpSs-2@Cluster implementation.

disconnected task graph, since the elements of the output vector are obtained from entirely separate computations. For this reason, there is a large difference between the Immediate Successor and Iteration Priority policies, with the Iteration Priority policy achieving about 35% lower performance than Immediate Successor on 128 nodes. This is because although Iteration Priority reliably achieves full parallelism until the end of the loop, it suffers from poor data locality resulting in lower cache utilization. The original OmpSs-@Cluster version scales to just 16 nodes, and it is already a factor of 2 times slower than fork–join MPI on 32 nodes. The multi-matvec benchmark has no inter-node communication, so the poor scaling of OmpSs-2@Cluster is due to the control message overhead for task offloading and dependency management, which is entirely eliminated by the distributed taskiter approach.

Figure 10c shows the initial overhead for multi-matmul, which has a maximum value of about 0.2 seconds on 128 nodes. Again, the overhead is expected to scale roughly linearly with the number of nodes, but the figure shows a high level of variability. As seen in Figure 10d, distributed taskiter achieves performance very close to the state-of-the-art TAMPI + OmpSs-2 implementation, with both scheduling policies. The results for MPI + OpenMP are less stable, due to differences in the task scheduling policy, which results in higher performance with 64 nodes and below, but a slightly lower performance on 128 nodes. The original OmpSs-2@Cluster version has roughly 50% lower performance than distributed taskiter on 64 nodes, after which point it fails to scale further in performance.

Figure 10e shows that the initial overhead for jacobi grows roughly quadratically from 2 to 64 nodes. This benchmark has all-to-all communication so the number of tasks and the number of accesses per task both grow roughly linearly in the number of cores, and they combine to cause the quadratic growth. On 64 nodes, we encounter unusual behavior in the MKL library, which exhibits degraded dgemv performance for small block sizes, so the optimal block size for 128 nodes is the same as that for 64 nodes. This explains why the quadratic overhead is interrupted at 64 nodes. The maximum overhead on 128 nodes is 60 seconds.

In Figure 10f, we see that the distributed taskiter version matches the fork–join MPI version, within 16.8%, on up to 64 nodes, but it drops to 29.9% below fork–join MPI on 128 nodes. There are two reasons for this. Firstly, the fork–join MPI version uses collective communication, whereas distributed taskiter always uses point-to-point communication. Secondly, the distributed taskiter version has asynchronous communication inside tasks, instead of fork–join parallelism, in this case, we see that the fork–join MPI + OmpSs-2, which also uses collective communications, matches fork–join MPI + OpenMP on all nodes and within 3.9% on 128 nodes. For this distributed taskiter benchmark, communication inside tasks incurs contention inside the MPI library. We see that the results match closely, within 7.8%, those for the asynchronous TAMPI + OmpSs-2 version on up to 64 nodes (which also has point-to-point communication inside tasks), while scaling beyond that, achieving 30.5% higher performance than TAMPI + OmpSs-2 on 128 nodes. Future work may investigate ways to implement collective communication and employ non-blocking collective MPI communication, although doing so risks unnecessary synchronization among processes, depending on the MPI implementation. In any case, the results already greatly outperform the original OmpSs-2@Cluster version.

Figure 10g shows the overhead for heat-gauss. Because of the 3D wavefront parallelism, it is most efficient to use the smallest block size that has acceptable performance overheads. The overhead is therefore roughly constant, rising from 1.8 seconds on 2 node to 2.6 seconds on 128 nodes. Figure 10h shows that the fork–join MPI + OpenMP version has poor performance, limited by the 2D wavefront parallelism inside each timestep. By enabling 3D wavefront parallelism, the asynchronous TAMPI + OmpSs-2 version achieves much higher performance, reaching performance 20.0 times faster than fork–join MPI + OpenMP on 128 nodes. The distributed taskiter version achieves similar performance, at 24.0 times faster than fork–join MPI + OpenMP on 128 nodes. In contrast, the OmpSs-2@Cluster implementation has even worse performance than fork–join MPI + OpenMP, being 7.0 times *slower*, on 128 nodes.

Figure 10i shows the overhead for heat-jacobi, the Jacobi version of the 2D heat-equation stencil computation. Each iteration is embarrassingly parallel, and the optimal block size is roughly constant from 1 to 128 nodes. The overhead is almost constant, rising to a maximum of just over 3.0 seconds on 128 nodes. Finally, in Figure 10j, all versions except the original OmpSs-2@Cluster implementation achieve similar scaling to at least 128 nodes. The distributed taskiter version is within 7.7% of the performance per iteration of fork–join MPI. Unlike the distributed taskiter variant, which creates a single set of tasks at the beginning and reuses them for subsequent iterations, the original OmpSs-2@Cluster re-creates tasks at each iteration. This leads to excessive memory consumption, causing it to run out of memory for this benchmark; hence, it is not included in the figures.

Overall, these results show that the initial overheads of distributed taskiter are acceptable, with a maximum of 1.1 seconds. The majority of this time is in the Step ③ graph translation described in Section 5.4, which is done by our implementation on one thread, but could easily be parallelized across all 56 threads in each process. The performance-per-iteration matches or exceeds fork–join MPI + OpenMP, with a drop between 16.8% 29.9% for the jacobi benchmark on 64 and 128 nodes. For heat-gauss, which benefits from asynchronous communication, the performance is similar to asynchronous TAMPI + OmpSs-2, at 20.0 times faster than fork–join MPI + OpenMP. In four out of the five benchmarks, the performance-per-iteration far exceeds that of the original OmpSs-2@Cluster, which is up to 7.7 times slower than fork–join MPI + OpenMP.

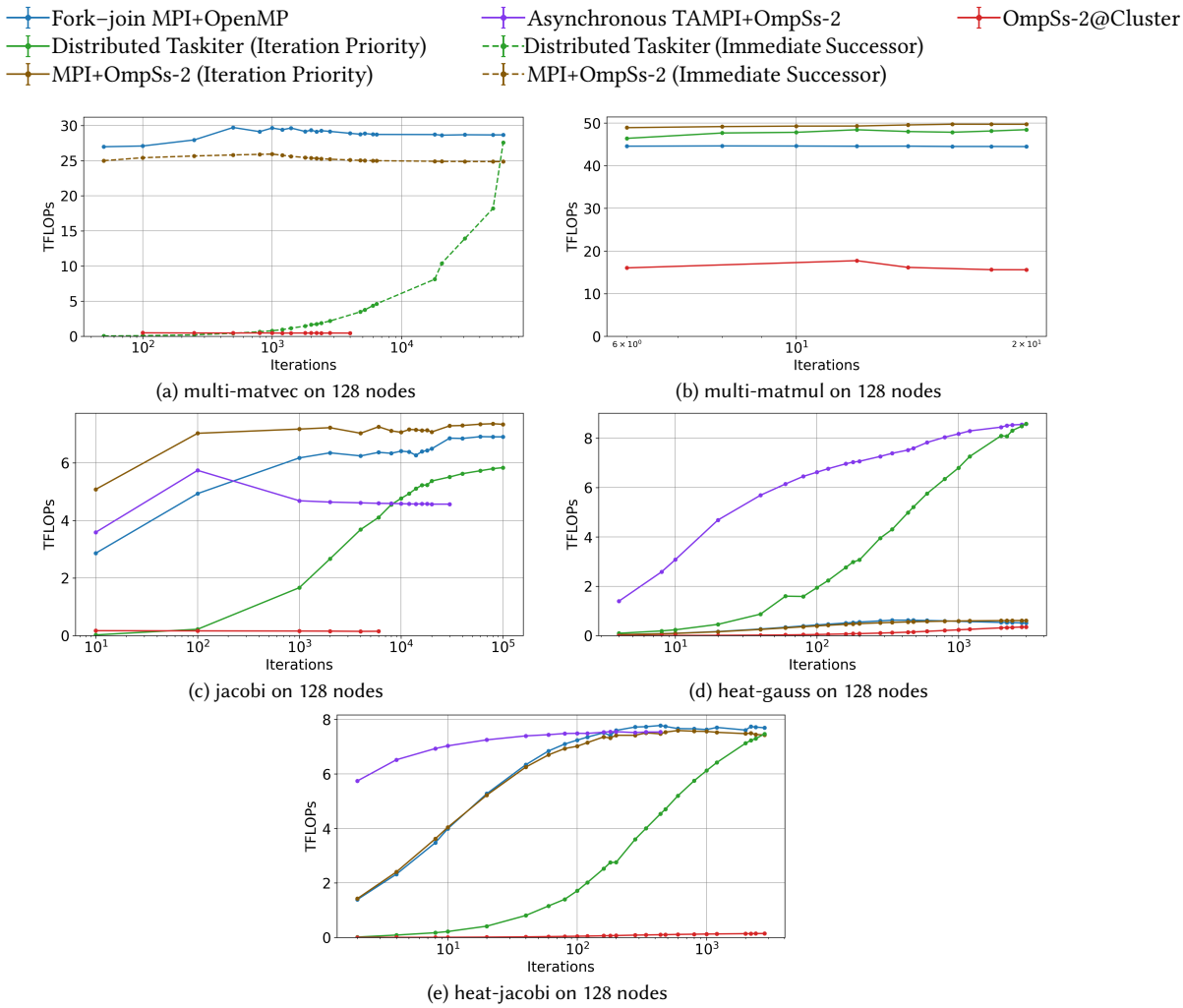
7.2 Effect of number of iterations on overall performance

Figure 11 shows the effect of the number of loop iterations on the overall performance. The five charts correspond to the five benchmarks run for up to 5 minutes worth of execution, in all cases executing on 128 nodes (256 processes). The x-axis is the number of iterations, on a log scale. The y-axis is the overall performance, in TFLOPS/s, which, unlike the right-hand column of Figure 10, includes the initial overhead. These results were observed afresh, not estimated synthetically by combining the left-hand and right-hand columns of Figure 10.

We see that, including the startup overhead, the distributed taskiter (Iteration Priority) version achieves higher performance for multi-matmul, and heat-gauss and heat-jacobi than the original OmpSs-2@Cluster version from the first iteration. For jacobi, the distributed taskiter (Iteration Priority) exceeds the original version from 10–100, while the distributed taskiter (Immediate Successor) exceeded multi-matvec from 100–1000 iterations.

The distributed taskiter (Immediate Successor) version has asymptotic performance to fork–join MPI + OpenMP and fork–join MPI + OmpSs-2 (Immediate Successor) variants, after 10^4 or more iterations for multi-matvec. For multi-matmul, however, the distributed taskiter (Iteration Priority) has higher performance from the first iteration (note the zoomed y-axis for multi-matmul, in order to expose the small differences between versions). For jacobi, distributed taskiter's (Iteration Priority) performance steadily increases up to about 18.3% below that of fork–join MPI + OpenMP around 10^5 iterations. For heat-gauss, the two versions that allow asynchronous overlap of timesteps, i.e., TAMPI + OmpSs-2 and distributed taskiter (Iteration Priority), have performance growing over the first approximately 1000 iterations, due to the increasing ability to overlap tasks from different timesteps through 3D wavefront parallelism. Finally, for heat-jacobi distributed taskiter (Iteration Priority) reaches close to the performance of MPI + OpenMP after about 1000 iterations.

It is important to note that the startup overhead has not been optimized in our current distributed taskiter implementation. As remarked above, most of the overhead is in the Step ③ graph translation described in Section 5.4. This is currently done by a single thread, but it could be parallelized across all 56 threads in each process.



(a) multi-matvec on 128 nodes

(b) multi-matmul on 128 nodes

(c) jacobi on 128 nodes

(d) heat-gauss on 128 nodes

(e) heat-jacobi on 128 nodes

Fig. 11. Overall performance, including all overheads, as a function of the number of iterations corresponding to approximately 5 minutes of computation. For multi-matmul, heat-gauss and heat-jacobi, the distributed taskiter (Iteration Priority) version outperforms the original OmpSs-2@Cluster version from the first iteration (unrolled by two).

7.3 Understanding initial overhead

Figure 12 plots the initial overhead in seconds on the y-axis versus the number of task access regions on the x-axis. The number of access regions is defined in Section 5.4.1 as the sum over tasks of the number of regions accessed by the task, where is it denoted R. Data points are shown for all benchmarks, numbers of nodes and a sweep of block sizes (not only the optimal block size used in Section 7.1). There is a small effect from the benchmark, in particular, forjacobi, which features all-to-all communication and more than average communication per access. Nevertheless, we see a strong linear relationship between the number of accesses and the initial overhead across four orders of magnitude, with just a few outliers, justifying the theoretical $O(R)$ complexity given in Section 5.4.1.

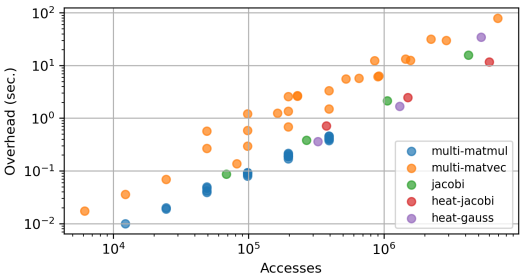


Fig. 12. Initial overhead as a function of the number of accesses.

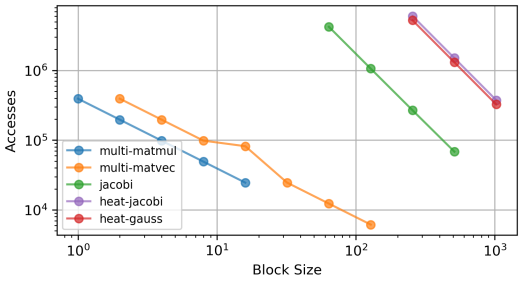


Fig. 13. Number of accesses depending on block size

Benchmark	Kernel			Control		
	Fork–join MPI+OpenMP	TAMPI+ OmpSs-2	Distributed taskiter	Fork–join MPI+OpenMP	TAMPI+ OmpSs-2	Distributed taskiter
multi-matvec	11	11 (+0%)	11 (+0%)	15	15 (+0%)	16 (+6.67%)
multi-matmul	18	18 (+0%)	18 (+0%)	15	15 (+0%)	16 (+6.67%)
jacobi	21	21 (+0%)	21 (+0%)	28	87 (+210.71%)	27 (-3.57%)
heat-gauss	40	40 (+0%)	40 (+0%)	45	60 (+33.33%)	25 (-44.44%)
heat-jacobi	40	40 (+0%)	40 (+0%)	54	76 (+40.74%)	30 (-44.44%)

Table 2. Number of lines of code for each variant of the benchmarks.

Figure 13 shows how the number of accesses, in turn, depends on the block size. The x-axis is the block size and the y-axis is the number of accesses, when executed on a single node. For multi-matvec and multi-matmul, which appear on top of each other since they have the same numbers of accesses, the number of accesses is a constant times the number of tasks, which is inversely proportional to the block size. The other benchmarks have their number of accesses inversely proportional to the square of the block size. For jacobi, this is because the number of tasks and the number of accesses per task are both inversely proportional to the block size. For heat-gauss and heat-jacobi, the number of accesses per task is fixed but the number of tasks is inversely proportional to the block size.

7.4 Quantifying productivity

Table 2 attempts to quantify the productivity of MPI+OpenMP, TAMPI+OmpSs-2 and distributed taskiter, using the commonly-used metric of lines of code. Quantifying the productivity of a programming system is a well known problem, and lines of code does not account for complexity, ease of understanding or the effects of varying programmer skill levels and preferences. It is, however, the best available metric because it is easily measured and understood.

The kernel code is isolated in a separate source file, and is the same for all implementations under study. The control code counts all source lines in the main loop of the benchmark, which makes a function call to the kernel code. [7] It excludes argument processing, memory allocation, initialization, and verification of the result. For multi-matvec and multi-matmul, which involve no communication among nodes, there is essentially no difference between the three approaches. For jacobi, the distributed taskiter version has a similar number of lines of code to MPI, which only requires a collective MPI_Allreduce, while the TAMPI+OmpSs-2 more than triples the number of lines of code (+210.7%). For heat-gauss and heat-jacobi, the asynchronous TAMPI+OmpSs-2 version increases the number of lines by at least a third, while the distributed taskiter version has just over half as many lines of code.

[7] There are some small differences between the numbers of lines of code in Table 2 and Figure 5, as the latter is reformatted for conciseness and clarity.

8 RELATED WORK

8.1 MPI, PGAS and hybrid approaches

Message Passing Interface (MPI) [38] is by far the most widely used standard for writing HPC applications, and it is well supported on all HPC systems. It is based on a distributed memory model with processes communicating via messages. Partitioned Global Address Space (PGAS) languages [34, 40, 57] and libraries [21, 29] provide a global address space, so that the processes access remote data directly, through language constructs or an API, rather than communicating via messages. This requires a more advanced understanding of memory consistency and synchronization. Both approaches, MPI and PGAS, place a high burden of data distribution, synchronization and load balancing on the programmer.

"MPI + X" models, which combine MPI with shared-memory parallelism via OpenMP [42], OpenACC [41], CUDA [36], or similar, have been under study for at least twenty years [31, 47]. Many applications use a fork–join approach, where processes alternate between sequential communication and parallel computation phases, which limits both inter- and intra-node parallelism. **Habanero-C MPI (HCMPI)** [23] addresses this limitation by automatically overlapping communication and computation, converting each MPI call into an asynchronous task. In general, attempts to move beyond a fork–join approach to overlap computation and communication introduce the risk of deadlock; for instance, blocking MPI primitives cannot be safely called within tasks. **Task-Aware MPI (TAMPI)** [51] provides a flexible solution to this problem, by integrating MPI with the task-based runtime, allowing tasks to safely and efficiently use MPI primitives (see Section 2.4). **StarPU-MPU's explicit MPI support** [9] defines a set of wrappers for MPI routines, which insert the proper data dependence edges between computation tasks and MPI requests [9]. Both these approaches achieve compelling performance and scalability that is hard to beat. In this paper, we compare our distributed taskiter approach with TAMPI, which has the benefit of using the same runtime system as used for the rest of the work. Despite their advantages, however, all MPI+X approaches inherit the fundamental complexities of MPI: programmers must manage data distribution, synchronization and load balancing. Additionally, they require the programmer to carefully divide the parallelism between shared and distributed memory models.

DASH [28] provides a C++ template library for distributed memory, which is based on tasks in a PGAS model. Each process concurrently creates its own task dependency graph. Tasks primarily access local memory, but they can also have dependencies on memory that is owned by another rank. Execution is divided into phases, and dependencies between tasks in different processes are only resolved at the boundaries between phases. Each rank communicates its non-local dependencies to the rank that owns the data, and the data values are exchanged as soon as they are available. This model supports tasks in a global memory space, but the programmer is still responsible for data distribution and load balancing. It is also necessary to divide the program into phases, during which there is no inter-node communication.

XcalableMP (XMP) [52] is a directive-based language that extends the C and Fortran languages for clusters, supporting explicit control of data distribution, communication, synchronization, and work mapping. Each node in a cluster executes the same main routine independently, while upon encountering an XMP construct, it is collectively executed by all nodes. Memory within a node is only accessible to that node, requiring explicit communication to access remote data. XMP supports global-view and local-view programming models. In the global-view model, the compiler determines the computation and data distribution based on user specified directives. The local-view model provides explicit control similar to MPI approaches.

Taskiter shares some similarities with **OpenMP Taskgraph** [60], a more general approach for recording and replaying sequences of tasks. Using Taskgraph, the user wraps a code region within a *taskgraph* directive, signaling that the enclosed code is the same every time it is encountered. When possible, the compiler uses static analysis to extract

the corresponding task graph. If static analysis is insufficient, the taskgraph is recorded during the first execution of the code. Subsequent executions then bypass task creation and dependency management overhead, since the taskgraph has already been captured. The key distinction between taskiter and taskgraph is that taskiter is specialized for iterative code. While this narrower focus reduces its applicability, it enables a highly compact DCTG representation to encode all iterations of the loop body. This specialization is central to the work in the current paper, which eliminates sequential control overhead and precomputes all MPI data transfers for all iterations of the loop body.

8.2 Distributed tasking

Distributed tasking approaches execute tasks with dependencies in a single task graph, which provides unambiguous dataflow semantics among all tasks of all processes. The task graph may exist only implicitly, based on a model of the structure of typical programs, but more commonly the model uses a Sequential Task Flow (STF) formulation. In the latter case, the task graph is constructed sequentially at runtime based on annotations or API calls. This may either be done concurrently on all processes, creating a duplicate task graph in each process, or the task graph may be created by a single process, which distributes work to the other processes. The whole unrolled graph for an STF program is built task-by-task, but it typically never exists in its completed form, since tasks are added (constructed) and removed (after execution) concurrently.

Implicit task graph creation: **PaRSEC** [18, 30] is a distributed task-based model designed for scalability on distributed heterogeneous distributed architectures. Originally developed to support the DPLASMA library for dense linear algebra, ParSEC, in its original formulation, builds a parameterized Directed Acyclic Graph (DAG) [24] that encodes task dependencies in an algebraic way in terms of the iteration variables. This approach eliminates the need to build the fully unrolled task graph, similarly to distributed taskiter. However, it introduces significant complexity in program development and is restricted to affine loops compatible with polyhedral analysis [20]. Our distributed taskiter approach also avoids constructing a full dependency graph, but it is designed specifically for iterative applications, offering a simpler and more accessible alternative. Unlike PaRSEC, distributed taskiter requires only a single pragma to identify such loops, reducing the complexity of implementation.

Concurrent and duplicated task graph creation: Several approaches build the same task graph, containing all top-level tasks and dependencies, concurrently on all processes. All processes independently determine the same deterministic mapping of tasks to rank and they execute only the tasks that are mapped to the current rank. Processes also insert appropriate send and receive primitives to pass data to and from tasks executed by other ranks.

StarPU-MPI [9] includes an extension of StarPU [6] to support distributed memory tasking using MPI, by mapping top-level tasks to nodes using an owner-computes model. **YarKhan**'s [59] extension of QUARK uses a deterministic mapping of task to rank based on data distribution, and it also uses MPI for communication. **TBLAS** [56] takes a similar approach to target clusters of CPUs with multiple GPUs. **OmpSs@cloudFPGA** [25] targets clusters of FPGAs with direct FPGA-to-FPGA communication and hardware acceleration to mitigate the cost of filtering task accesses. **DuctTeip** builds hierarchical data structures and task graphs, mitigating the sequential bottleneck through a task nesting approach. **PaRSEC** [18, 30] also supports Dynamic Task Discovery (DTD) as an alternative to the Parameterized Task Graph (PTG). DTD constructs a general task graph from a sequential program, unrolling the full task dependency graph on each process. This is similar to the previously-described approaches, and it suffers from the same bottleneck and flexibility issues.

In all the above approaches, all ranks must independently determine the same mapping of task to rank, which makes it impossible to dynamically load balance. Every rank has to check the dependencies of every top-level task

in every iteration, which limits the scalability for fine- and medium-grained tasks. Our approach has the advantage that it does not require task nesting with the possibility to support it (if needed), it is compatible with partitioning and re-partitioning of the cyclic task graph, and the entire overhead to build and manage the task graph and insert communication is amortized across all loop iterations. Finally, it interoperates with the fully-general OmpSs-2@Cluster approach, which starts from a single sequential thread.

Sequential task graph creation: Other approaches build the top-level task graph on a single node, with task offloading to other nodes. **OmpSs-1@Cluster** [19] and its successor **OmpSs-2@Cluster** (Section 2.2) extend BSC's OmpSs programming model to support distributed memory clusters. Both create the dependency graph on a single core, although OmpSs-2@Cluster has improved support for task nesting as a way to parallelize the creation of tasks on multiple ranks to reduce the pressure on the first rank. Our approach is compatible with OmpSs-2@Cluster, but it avoids all of the control message overhead inside the timesteps of iterative applications (see Section 2.2.2). We compare our results against OmpSs-2@Cluster in detail in Section 7,and demonstrate that while the existing OmpSs-2@Cluster approach is a viable alternative to MPI+OpenMP on up to 4 or 8 nodes, our approach is close to MPI + OpenMP on up to at least 128 nodes.

Legion [17] is a framework for parallel tasking computations on distributed heterogeneous systems. Execution also begins on a single rank, and tasks are offloaded to other ranks. It supports task nesting and adopts a data-centric approach, where developers describe the structure and properties of data, so that the scheduler can optimize data locality. Programs can either use Legion's native C++ API or the high-productivity **Regent** language [54]. It has similar disadvantages to OmpSs-2@Cluster, in terms of scalability and the need to use task nesting to get good performance.

8.3 Other approaches

Charm++ [43] is an asynchronous execution model for HPC, based on migratable objects known as "chares". Chares communicate by exchanging messages, resulting in a form of concurrent and asynchronous execution that has some similarities to task execution without dependencies. **HPX** [32] is a C++ library that supports parallel computations using an interface that aims to be compatible with the C++ Standard Template Library (STL). It adopts an asynchronous and distributed task-based model that is expressed using futures, and which supports data dependencies among futures. **X10** [22] is an object-oriented programming language for high-productivity programming that spawns asynchronous computations, with the programmer responsible for PGAS data distribution.

Chapel [1, 58] is a high productivity HPC programming language, which allows asynchronous tasks to be spawned and executed on other distributed-memory nodes. Chapel's *sync* qualifier can block tasks until the necessary data is ready, allowing for coordinated execution order among tasks. Unlike other tasking models, a task graph is not created in advance of task execution.

OMPC [39, 61] extends the LLVM OpenMP implementation of the target library with new target for offloading tasks, and introduces the concept of a "remote device" for offloading tasks to remote nodes. Rosso et al. extends the **OMPC** [61] MPI plugin [48] to support OpenMP offloading across distributed systems with FPGAs. Similarly [44] implemented a plugin on top of the existing target construct of the LLVM/OpenMP implementation for offloading to CPUs and GPUs using RPC on top of UCX as the communication layer. This plugin was refactored later by [53] to use MPI instead of UCX as the communication layer. It uses an offload model similar OmpSs-2@Cluster, Legion and Regent, and therefore suffers from the same overheads due to sequential task creation and offloading.

8.4 Scripting and workflows

Many scripting and workflow frameworks also adopt a distributed tasking approach with a directed acyclic graph of tasks and dependencies. **COMPSs** [35] is a Java, C/C++ and Python framework to run parallel applications on clusters, clouds and containerized platforms. It has a sequential task-based model similar to OmpSs, but dependencies are tracked through files or objects rather than the virtual address space. **Pegasus** [26] is another workflow management system that uses a DAG of tasks and dependencies. **GPI-Space** [49] is a fault-tolerant execution platform for data-intensive applications. It supports coarse-grained tasks that decouples the domain user from the parallel execution of the problem. In all these approaches, the task granularity is much coarser than that targeted by our approach, with individual tasks having duration up to hours or days. It is viable for a single master node to manage all task scheduling and data transfers.

9 CONCLUSIONS

Despite being very productive, distributed Sequential Task Flow (STF) models suffer from limited performance and scalability for fine- and medium-grained tasks. This paper presents an extension to OmpSs-2@Cluster that addresses this issue for applications with common iterative patterns, while maintaining the productive OmpSs-2@Cluster programming model. While the existing OmpSs-2@Cluster implementation scales to only about 16 nodes with medium-scale tasks and problem sizes that scale to about 32 nodes, our approach scales similarly to MPI+OpenMP on up to 128 nodes, with a maximum slowdown of 30%, compared with fork–join MPI + OpenMP. When the application has the potential to overlap iterations, for example the 2D heat equation stencil calculation with Gauss–Seidel updates, our approach discovers significantly more parallelism than fork–join MPI + OpenMP. This results in up to 24.0 times higher performance on 128 nodes, which is on-a-par with state-of-the-art asynchronous TAMPI + OmpSs-2. As such, the model combines the productivity of STF models with the performance of state-of-the-art MPI+X approaches, by exploiting the iterative nature of scientific applications. It also avoids the synchronization and deadlock issues of an MPI+X approach. Future work will build on this foundation through automatic partitioning, optimizations in the startup and potentially use of collectives, as well as repartitioning for dynamic load balance and malleability.

10 ACKNOWLEDGEMENTS

This research has received funding from the European Union's Horizon 2020/EuroHPC research and innovation programme under grant agreement No 955606 (DEEP-SEA), and PN038700 (Zettascale Laboratory). It is also supported by the Spanish State Research Agency - Ministry of Science and Innovation under contract PID2019-107255GB-C21/-MCIN/AEI/10.13039/501100011033 and Ramon y Cajal fellowship RYC2018-025628-I/MCIN/AEI/10.13039/501100011033 and by "ESF Investing in your future", as well as by the *Generalitat de Catalunya* (2017-SGR-1414).

REFERENCES

[1] 2022. *The Chapel Parallel Programming Language.* https://chapel-lang.org/ Accessed: 2022-04-04.

[2] 2024. Clang: a C language family frontend for LLVM. https://clang.llvm.org/ Accessed: 2024-01-18.

[3] 2024. The LLVM Compiler Infrastructure. https://llvm.org/ Accessed: 2024-01-18.

[4] Jimmy Aguilar Mena, Omar Shaaban, Vicenç Beltran, Paul Carpenter, Eduard Ayguadé, and Jesus Labarta. 2022. OmpSs-2@Cluster: Distributed memory execution of nested OpenMP-style tasks. In *European Conference on Parallel Processing.* https://doi.org/10.1007/978-3-031-12597-3_20

[5] Jimmy Aguilar Mena, Omar Shaaban, Victor Lopez, Marta Garcia, Paul Carpenter, Eduard Ayguadé, and Jesus Labarta. 2022. Transparent load balancing of MPI programs using OmpSs-2@Cluster and DLB. In *51st International Conference on Parallel Processing (ICPP).* https://doi.org/10.1145/3545008.3545045

[6] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Paul Thibault. 2017. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems*

(2017). https://doi.org/10.1109/TPDS.2017.2766064

[7] David Álvarez and Vicenç Beltran. 2023. Optimizing Iterative Data-flow Scientific Applications using Directed Cyclic Graphs. *IEEE access* (2023). https://doi.org/10.1109/ACCESS.2023.3269902

[8] David Álvarez, Kevin Sala, Marcos Maroñas, Aleix Roca, and Vincenç Beltran. 2021. Advanced Synchronization Techniques for Task-Based Runtime Systems. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA. https://doi.org/10.1145/3437801.3441601

[9] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. 2012. StarPU-MPI: Task programming over clusters of machines enhanced with accelerators. In *European MPI Users' Group Meeting*. https://doi.org/10.1007/978-3-642-33518-1_40

[10] Barcelona Supercomputing Center. 2021. *Influence in OpenMP - OmpSs-2 specification.* https://pm.bsc.es/ftp/ompss-2/doc/spec/introduction

[11] Barcelona Supercomputing Center. 2021. *Nanos6.* https://github.com/bsc-pm/nanos6

[12] Barcelona Supercomputing Center. 2021. *OmpSs-2 Specification.* https://pm.bsc.es/ftp/ompss-2/doc/spec/

[13] Barcelona Supercomputing Center. 2021. *Taskiter.* https://github.com/bsc-pm/ompss-2-cluster-releases#taskiter

[14] Barcelona Supercomputing Center. 2022. OmpSs-2@Cluster releases. https://github.com/bsc-pm/ompss-2-cluster-releases

[15] Barcelona Supercomputing Center. 2024. MareNostrum 5 System Architecture. https://www.bsc.es/marenostrum/marenostrum-5

[16] Barcelona Supercomputing Center. 2024. *OmpSs-2 LLVM Compiler Infrastructure.* https://github.com/bsc-pm/llvm

[17] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* https://doi.org/10.1109/SC.2012.71

[18] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pitior Luszczek, and Jack Dongarra. 2012. *Dense linear algebra on distributed heterogeneous hardware with a symbolic DAG approach.* Technical Report. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States). https://www.osti.gov/servlets/purl/1173290

[19] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta. 2012. Productive Programming of GPU Clusters with OmpSs. In *IEEE 26th International Parallel and Distributed Processing Symposium.* https://doi.org/10.1109/IPDPS.2012.58

[20] Paul Cardosi and Bérenger Bramas. 2023. Specx: a C++ task-based runtime system for heterogeneous distributed architectures. *arXiv preprint arXiv:2308.15964* (2023).

[21] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model.* https://doi.org/10.1145/2020373.2020375

[22] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA). https://doi.org/10.1145/1094811.1094852

[23] Sanjay Chatterjee, Sagnak Tasırlar, Zoran Budimlic, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. 2013. Integrating asynchronous task parallelism with MPI. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE. https://doi.org/10.1109/IPDPS.2013.78

[24] M. Cosnard and M. Loi. 1995. Automatic task graph generation techniques. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences.* https://doi.org/10.1109/HICSS.1995.375471

[25] Juan Miguel de Haro, Rubén Cano, Carlos Álvarez, Daniel Jiménez-González, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, Francois Abel, Burkhard Ringlein, and Beat Weiss. 2022. OmpSs@cloudFPGA: An FPGA Task-Based Programming Model with Message Passing. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* https://doi.org/10.1109/IPDPS53621.2022.00085

[26] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Berriman, John Good, Anastasia Laity, and Daniel S. Katz. 2005. Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming* 3 (2005). https://doi.org/10.1155/2005/128026

[27] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters* 02 (2011). https://doi.org/10.1142/S0129626411000151

[28] Karl Fürlinger, José Gracia, Andreas Knüpfer, Tobias Fuchs, Denis Hünich, Pascal Jungblut, Roger Kowalewski, and Joseph Schuchart. 2020. DASH: Distributed Data Structures and Parallel Algorithms in a Global Address Space. In *Software for Exascale Computing-SPPEXA 2016-2019.* https://doi.org/10.1007/978-3-030-47956-5_6

[29] Daniel Grünewald and Christian Simmendinger. 2013. The GASPI API specification and its implementation GPI 2.0. In *7th International Conference on PGAS Programming Models.*

[30] Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. 2017. Dynamic task discovery in PaRSEC: a data-flow task-based runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems.* https://doi.org/10.1145/3148226.3148233

[31] Gabriele Jost, Hao-Qiang Jin, Ferhat F Hatay, et al. 2003. Comparing the OpenMP, MPI, and hybrid programming paradigm on an SMP cluster. In *European Workshop on OpenMP and Applications 2003.* https://ntrs.nasa.gov/citations/20030107321

[32] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A task based programming model in a global address space. In *8th International Conference on Partitioned Global Address Space Programming Models.* https://doi.org/10.13140/2.1.2635.5204

[33] Jannis Klinkenberg, Philipp Samfass, Michael Bader, Christian Terboven, and Matthias Müller. 2019. CHAMELEON: Reactive load balancing for hybrid MPI+OpenMP task-parallel applications. *J. Parallel and Distrib. Comput.* (2019). https://doi.org/10.1016/j.jpdc.2019.12.005

[34] Jinpil Lee, Minh Tuan Tran, Tetsuya Odajima, Taisuke Boku, and Mitsuhisa Sato. 2011. An extension of XcalableMP PGAS lanaguage for multi-node GPU clusters. In *Proceedings of the 2011 International Conference on Parallel Processing* (Bordeaux, France). Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-29737-3_48

[35] Francesc Lordan, Enric Tejedor, Jorge Ejarque, Roger Rafanell, Javier Alvarez, Fabrizio Marozzo, Daniele Lezzi, Raül Sirvent, Domenico Talia, and Rosa M Badia. 2014. ServiceSs: An interoperable programming framework for the cloud. *Journal of grid computing* 1 (2014). https://doi.org/10.1007/s10723-013-9272-5

[36] David Luebke. 2008. CUDA: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*. IEEE. https://doi.org/10.1109/ISBI.2008.4541126

[37] Jimmy Aguilar Mena. 2022. *Methodology for malleable applications on distributed memory systems*. Ph. D. Dissertation. Universitat Politècnica de Catalunya. http://dx.doi.org/10.5821/dissertation-2117-380814

[38] MPI Forum. 2023. MPI Documents. https://www.mpi-forum.org/docs/

[39] Rémy Neveu, Rodrigo Ceccato, Gustavo Leite, Guido Araujo, Jose M Monsalve Diaz, and Hervé Yviquel. 2024. Towards an Optimized Heterogeneous Distributed Task Scheduler in OpenMP Cluster. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[40] Robert W Numrich and John Reid. 1998. Co-Array Fortran for parallel programming. In *ACM SIGPLAN Fortran Forum*. ACM New York, NY, USA. https://doi.org/10.1145/289918.289920

[41] OpenACC Organization. 2011. OpenACC: Directives for Accelerators. http://www.openacc-standard.org

[42] OpenMP Architecture Review Board. 2021. OpenMP Application Programming Interface, Version 5.2. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf Accessed: 2022-04-19.

[43] Parallel Programming Lab, Dept of Computer Science, University of Illinois. 2023. *Charm++ Documentation*. https://charm.readthedocs.io/en/latest

[44] Atmn Patel and Johannes Doerfert. 2022. Remote OpenMP Offloading. In *High Performance Computing*, Ana-Lucia Varbanescu, Abhinav Bhatele, Piotr Luszczek, and Baboulin Marc (Eds.). Springer International Publishing, Cham.

[45] J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé. 2017. Improving the Integration of Task Nesting and Dependencies in OpenMP. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. https://doi.org/10.1109/IPDPS.2017.69

[46] Josep Pérez, Rosa M. Badia, and Jesús Labarta. 2008. A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures. *IEEE International Conference on Cluster Computing, ICCC*. https://doi.org/10.1109/CLUSTR.2008.4663765

[47] Rolf Rabenseifner and Gerhard Wellein. 2005. Comparison of parallel programming models on clusters of SMP nodes. In *Modeling, Simulation and Optimization of Complex Processes: Proceedings of the International Conference on High Performance Scientific Computing, March 10–14, 2003, Hanoi, Vietnam*. Springer. https://doi.org/10.1007/3-540-27170-8_31

[48] Pedro Henrique Rosso, Lucian Petrica, Nusrat Jahan Lisa, Marcio Pereira, Sandro Rigo, Hervé Yviquel, Vanderlei Bonato, Emilio Francesquini, and Guido Araujo. 2024. Integrating Multi-FPGA Acceleration to OpenMP Distributed Computing. In *Advancing OpenMP for Future Accelerators*, Alexis Espinosa, Michael Klemm, Bronis R. de Supinski, Maciej Cytowski, and Jannis Klinkenberg (Eds.).

[49] Tiberiu Rotaru, Mirko Rahn, and Franz-Josef Pfreundt. 2014. MapReduce in GPI-Space. In *Euro-Par 2013: Parallel Processing Workshops*. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-54420-0_5

[50] Kevin Sala, Sandra Macià, and Vicenç Beltran. 2021. Combining One-Sided Communications with Task-Based Programming Models. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. https://doi.org/10.1109/Cluster48925.2021.00024

[51] Kevin Sala, Xavier Teruel, Josep M Perez, Antonio J Peña, Vicenç Beltran, and Jesus Labarta. 2019. Integrating blocking and non-blocking MPI primitives with task-based programming models. *Parallel Comput.* (2019). https://doi.org/10.1016/j.parco.2018.12.008

[52] Mitsuhisa Sato. 2021. *XcalableMP PGAS Programming Language: From Programming Model to Applications*. Springer Nature.

[53] Baodi Shan, Mauricio Araya-Polo, Abid M. Malik, and Barbara Chapman. 2023. MPI-based Remote OpenMP Offloading: A More Efficient and Easy-to-use Implementation. In *Proceedings of the 14th International Workshop on Programming Models and Applications for Multicores and Manycores*. NY, USA. https://doi.org/10.1145/3582514.3582519

[54] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. 2015. Regent: a high-productivity programming language for HPC with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. https://doi.org/10.1145/2807591.2807629

[55] Lorna Smith and Mark Bull. 2001. Development of mixed mode MPI / OpenMP applications. *Sci. Program.* (2001).

[56] Fengguang Song and Jack Dongarra. 2012. A scalable framework for heterogeneous GPU-based clusters. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*.

[57] UPC Consortium. 2024. Berkeley UPC – Unified Parallel C. https://upc.lbl.gov/

[58] Michele Weiland. 2007. Chapel, Fortress and X10: novel languages for HPC. *EPCC, The University of Edinburgh, Tech. Rep. HPCxTR0706* (2007).

[59] Asim YarKhan. 2012. *Dynamic task execution on shared and distributed memory architectures*. Ph. D. Dissertation. University of Tennessee.

[60] Chenle Yu, Sara Royuela, and Eduardo Quinones. 2023. Taskgraph: A low contention openmp tasking framework. *IEEE Transactions on Parallel and Distributed Systems* (2023).

[61] Hervé Yviquel, Marcio Pereira, Emílio Francesquini, Guilherme Valarini, Gustavo Leite, Pedro Rosso, Rodrigo Ceccato, Carla Cusihualpa, Vitoria Dias, Sandro Rigo, Alan Souza, and Guido Araujo. 2023. The OpenMP Cluster Programming Model. In *Workshop Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France). https://doi.org/10.1145/3547276.3548444