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

Omar Shaaban[a,*], Juliette Fournis d'Albiat[a], Isabel Piedrahita[a], Vicenç Beltran[a], Paul Carpenter[a], Eduard Ayguadé[a], Jesus Labarta[a]

[a]*Barcelona Supercomputing Center, Placa Eusebi Guell, 1–3, 08034, Barcelona, Spain*

Abstract

Distributed tasking approaches, such as OmpSs-2@Cluster, StarPU-MPI, PaRSEC and others, express an HPC application as a graph of tasks with dependencies. The single task graph avoids the synchronization and deadlock issues of an MPI + X approach and it unifies the representation of parallelism across CPU cores, accelerators, and distributed-memory nodes. Most models create the task graph sequentially, which provides a clear and familiar programming interface, simplifying development and maintenance and facilitating the porting of existing codes. The downside is the sequential bottleneck, which limits performance and scalability. Unless the tasks are very large, distributed sequential task graph approaches are not yet a viable alternative to MPI + X.

A large class of applications implement iterative methods or multi-step simulations, which create the same directed acyclic graph of tasks on each timestep. We exploit this inherent structure in a sequentially-constructed distributed tasking model, in order to eliminate the sequential bottleneck and control messages, while retaining the simplicity and productivity of the approach. This is done by taking advantage of the recently-proposed taskiter directive to create a single iteration as a cyclic graph and precompute the MPI transfers, reusing this representation for all subsequent iterations. 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 32 nodes of the MareNostrum 4 supercomputer. For applications with fork–join parallelism, our approach has performance similar to fork–join MPI + OpenMP. It is therefore a viable productive alternative to MPI + OpenMP, unlike the existing OmpSs-2@Cluster implementation, which is up to 15 times slower than MPI + OpenMP. Our method translates the body of the loop into asynchronous tasks with non-blocking communication, which in some cases exposes dramatically more parallelism than fork–join MPI + OpenMP. For a 2D Gauss–Seidel stencil computation, for example, our approach enables a 3D wavefront computation, giving performance up to 11 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. Introduction

Distributed-memory tasking approaches, such as OmpSs-2@Cluster [3], StarPU-MPI [8], PaRSEC [29], OMPC [56] and others [32], express an HPC application using a single graph of tasks with dependencies. These tasks are mapped to processes and executed concurrently across all available compute nodes. Distributed tasking approaches avoid the synchronization and deadlock issues of an MPI+X approach [48, 47, 51], and they can be extended to naturally support transparent dynamic load balancing [4, 32] and both core- and node-level malleability [36]. While some models use an implicit task graph, such as the Parameterized Task Graph (PTG) originally used for PaRSEC [23], the majority use a Sequential Task Graph (STG) model, where the task graph is constructed sequentially. The STF approach provides a clear and familiar meaning to the program, which simplifies development and maintenance, as well as 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 [36], while other STF approaches that create tasks in parallel, such as OmpSs@cloudFPGA [24] and StarPU-MPI achieve somewhat better scalability, but they are still ultimately limited by the sequential filtering of task dependencies. Moreover, all nodes need to independently agree on the same mapping of tasks to nodes. This static or deterministic allocation of tasks to nodes makes it impossible to transparently balance the load across nodes.

At the same time, many HPC applications implement iterative methods or multi-step simulations, which create the same directed acyclic graph (DAG) of tasks on each timestep. A recent paper proposed the taskiter directive [6], which gives application programmers a way to express that a specific loop creates the same pattern of tasks on each iteration. This information allows the runtime to build the DAG for a single iteration and link it into a cyclic graph that describes the overall computation, thereby amortizing the costs of task creation, scheduling

*Corresponding author

Email addresses: omar.ibrahim@bsc.es (Omar Shaaban), juliette.fournis@bsc.es (Juliette Fournis d'Albiat), isabel.piedrahita@bsc.es (Isabel Piedrahita), vicenc.beltran@bsc.es (Vicenç Beltran), paul.carpenter@bsc.es (Paul Carpenter), eduard.ayguade@bsc.es (Eduard Ayguadé), jesus.labarta@bsc.es (Jesus Labarta)

and dependency management across all iterations of the loop.

Taskiter was proposed for SMPs, but it is a particularly good fit for distributed tasking, for two main reasons. Firstly, as the number of nodes grows, the number of tasks normally 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 wall-clock 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 total 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, these data transfers need to be computed dynamically, as the graph is built, which results 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 translates the body of the loop into asynchronous tasks and non-blocking MPI communication. This fully eliminates the control message overhead. By performing the communication inside asynchronous tasks, it naturally overlaps communication and computation, in some cases exposing dramatically more parallelism than the typical fork–join MPI + OpenMP approach.

We implemented it in the Nanos6@Cluster runtime system and evaluated it using five benchmarks on up to 32 nodes of the MareNostrum 4 supercomputer. Our results improve on the existing OmpSs-2@Cluster approach, which, for 64 processes on 32 nodes, is slower than MPI + OpenMP, by between 6 and 15 times for three of the five benchmarks. In contrast, our maximum slowdown with respect to MPI + OpenMP is just 15%. For four of the five benchmarks, our performance either exceeds that of MPI + OpenMP or is within 5.0%. For the 2D heat equation stencil calculation, discussed in Section 3, which has the potential for 3D wavefront parallelism, we achieve 11 times higher performance than fork–join MPI + OpenMP, on-a-par with state-of-the-art asynchronous Task Aware MPI (TAMPI) + OmpSs-2. 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 the current OmpSs-2@Cluster as an efficient and portable distributed task programming model for iterative applications.
- We describe the runtime implementation and related optimizations.
- We implement our model in the Nanos6@Cluster runtime system and provide experimental results on up to 32 nodes of the MareNostrum 4 supercomputer.
- We demonstrate that while the existing OmpSs-2@Cluster approach is only viable for up to 4 or 8 nodes, our approach is on-a-par with an asynchronous TAMPI + tasks hand-optimized implementation up to at least 32 nodes.

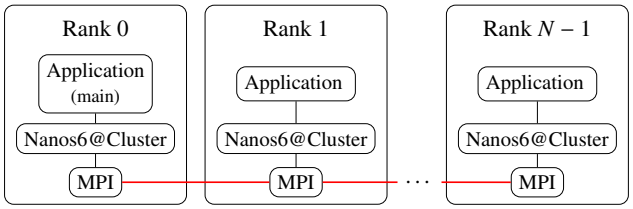


Figure 1: OmpSs-2@Cluster architecture in which each rank is a peer. The main function is executed as a task on Rank 0. All other tasks may be offloaded for execution by any other rank.

The rest of this paper is structured as follows. Section 2 is a summary of the relevant background in OmpSs-2, OmpSs-2@Cluster, Taskiter and Task-Aware MPI (TAMPI). 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 [26] programming model. It is open source and used as a research platform to develop and evaluate ideas that may be proposed for future standardization in OpenMP [40, 10]. OmpSs-2 is based on directives and it enables parallelism in a dataflow way [44]. OmpSs-2 is similar to OpenMP except that it uses a thread-pool execution model, where workers are created at startup, rather than the fork–join model of OpenMP, which creates threads only for the parallel sections. It targets heterogeneous architectures through native kernels, and uses asynchronous task parallelism as the main mechanism for concurrency. Task data accesses are a unified way to express task ordering dependencies, optimize data locality and determine data transfers to and from accelerators. The reference implementation of OmpSs-2 comprises the OmpSs-2 compiler [15], which is based on LLVM [2]/Clang [1], and the Nanos6 [11] runtime.

OmpSs-2 extends OmpSs to improve task nesting and fine-grained dependencies across nesting levels [42, 7]. It adds weak access types: *weakin*, *weakout* and *weakinout*, which specify that a task does not directly access the corresponding region, but its subtasks may read or modify the data. Weak accesses do not delay task execution, but they link between dependency domains, to support fine-grained task ordering and enable the passing of data location information to and from the subtasks.

2.2. OmpSs-2@Cluster

OmpSs-2@Cluster is the extension of OmpSs-2 to support task offloading among nodes [3, 14]. As an alternative to MPI + OmpSs-2, it scales to about 16 nodes [3], depending on the algorithm and problem size. It supports active malleability, interacting with the job scheduler to request or release compute nodes, and then making use of these resources in a way that

is transparent to the programmer [36]. It can also be used to provide multi-node dynamic load balancing for MPI + OmpSs-2 programs [4], using its ability to offload tasks to ensure that each node has an equal amount of work.

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. The auto keyword allows tasks to access data whose address is not known at task creation time [49]. Application data is allocated either as *local memory*, which is accessible to the (parent) task that performs the allocation, as well as its subtasks, and copied back on a taskwait of the parent task, or as *distributed memory*, which is accessible only by subtasks and not copied back on a taskwait of the parent.[1]

2.2.1. Tasks, offloading and scheduling

The main function executes as a task on the first process, Rank 0. All other tasks are created as subtasks of their parent, initially on the node that executes the parent task. Top-level tasks are therefore initially created on Rank 0. If the task is to be executed locally, then it is passed to the host scheduler when it becomes ready, in the normal way. Otherwise, since the task will be executed remotely, an MPI message is sent to the remote rank. On receipt, the remote rank creates a copy of the task, which is scheduled by the host scheduler on the remote rank. Scheduling is therefore done at two levels: the cluster scheduler of the parent's rank maps the task to the execution rank and the host scheduler on the execution rank schedules the task to run on an available core.

Optimized OmpSs-2@Cluster programs typically have two levels of nested tasks. The top level has one offloadable task per process, in order to distribute the work across processes, and the second level has a small number of non-offloadable tasks per core, in order to distribute the work across the cores on that process. This approach mitigates the sequential task creation and offloading bottleneck, at the cost of some programmer complexity, and it is responsible for a good part of the scalability of OmpSs-2@Cluster to about 16 nodes [3]. Since the top-level tasks do not themselves perform computation, they can execute, and create their subtasks, before the data is ready. This is made possible using weak accesses on the top-level tasks (see Section 2.1). All baseline OmpSs-2@Cluster benchmarks (without taskiter) in this paper therefore use two levels of nested tasks (see Section 6).

2.2.2. Control messages and global write ordering

Figure 2 is an example that shows the control messages that OmpSs-2@Cluster uses to offload tasks and enforce dependencies. Figure 2 is an OmpSs-2@Cluster program that creates two tasks, A and B, and offloads them from Rank 0 to Rank 1

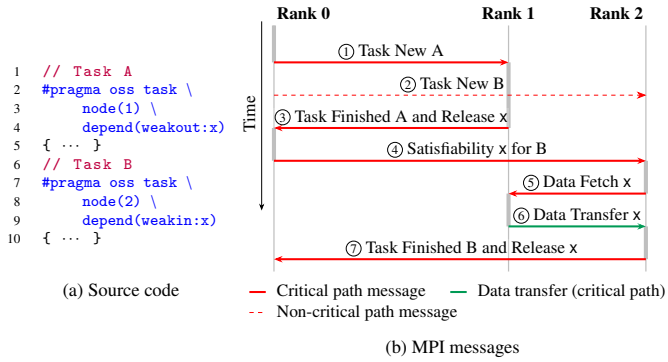


Figure 2: Large number of MPI messages for OmpSs-2@Cluster. Offloading and executing these two tasks requires one data transfer message and six control messages.

and Rank 2, respectively. These tasks have weak accesses, as discussed above in Section 2.2.1. The task execution rank is specified for concreteness in these examples, by overruling the cluster scheduler using the node() clause. Figure 2(b) shows the sequence of MPI messages, from top to bottom, involved in the execution. First, Task A and Task B are offloaded to Rank 1 and Rank 2, respectively with two consecutive Task New messages. When Task A finishes on Rank 1, a Task Finished message is sent back to Rank 0. This message releases the output value of x and identifies the location of the latest version of the data. In turn, Rank 0 sends a Satisfiability message to Rank 2, which passes global write permission and the current location of x. Since the data is on Rank 1, Rank 2 sends a Data Fetch control message to Rank 1, which responds by posting a point-to-point data transfer containing the data. Finally, when task B completes, Rank 2 sends a Task Finished message to Rank 0. The runtimes collectively enforce a global ordering of writes, and they send and receive a total of seven messages, all except one (to offload B) on the critical path. Only one of these messages carries the actual data.

2.3. Taskiter

The *taskiter* construct was recently proposed [6] for OmpSs-2 and OpenMP. It declares that each iteration of the associated loop creates the same dependency graph of tasks and accesses. Moreover, the program remains valid if the code inside the loop body but outside any task is executed just once.

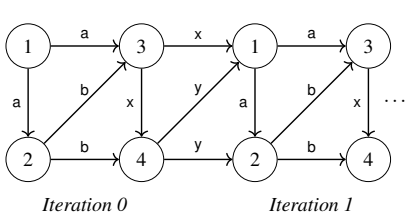
An example program using taskiter is shown in Figure 3a, where the only modification to take advantage of taskiter is the pragma annotation on line 1. Figure 3b shows the regular task graph. When using taskiter, however, the runtime only executes one iteration of the taskiter's loop, creating the dependency graph for a single iteration. It then converts the graph into the directed cyclic task graph (DCTG) shown in Figure 3c.[2] In this figure, the non-cyclic edges, between tasks in the same

[1] Distributed data can be copied back to main using taskwait on.

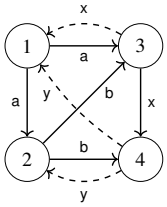
[2] The graph created by the runtime may have additional edges for the write-after-write (WaW) dependencies between consecutive iterations of the same task. These WaW dependencies are implied by existing paths in the graph, and have been omitted from both subfigures.

```
1  #pragma omp taskiter
2  for(int it=0; it<NUM_ITERATIONS; it++) {
3       // Task 1
4       #pragma omp task depend(in:x,y) depend(out:a)
5       { ··· }
6       // Task 2
7       #pragma omp task depend(in:a,y) depend(out:b)
8       { ··· }
9       // Task 3
10      #pragma omp task depend(in:a,b) depend(out:x)
11      { ··· }
12      // Task 4
13      #pragma omp task depend(in:x,b) depend(out:y)
14      { ··· }
15 }
```

(a) Example OpenMP program using taskiter construct



(b) Normal task dependence graph without taskiter



(c) DCTG with taskiter

Figure 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).

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. This also reduces the impact on execution time of the more powerful but expensive fragmented regions dependency system [43], since dependency system operations are only performed for the tasks in a single iteration.

As normal, the tasks can create subtasks. Although the first-level task graph must be the same each iteration, their subtasks, if any, may be different in each iteration, allowing irregularity between iterations at deeper nesting levels. The optional unroll(n) clause enables taskiter to support a loop whose task graph repeats every n iterations.

Taskiter does not require the taskiter to have a run-time constant number of iterations. If the number of iterations cannot be determined by the compiler, then the compiler 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 strictly separates the computation ("X") and communication (MPI) through additional synchronization, which, as we confirm in our results, hinders 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.

```
1  double x[10];
2  ···
3  #pragma omp task depend(out:x)
4  {
5    MPI_Request request;
6    MPI_Irecv(&x, 10, MPI_DOUBLE, other_rank, tag,
           MPI_COMM_WORLD, &request);
7    TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
8  }
9  #pragma omp task depend(in:x)
10 {
11   ···
12 }
```

Figure 4: Example OpenMP program using Task-aware MPI's (TAMPI's) non-blocking communication. The call to TAMPI_Iwait makes the completion of the MPI call on line 6 visible to the runtime's dependency system, simplifying the code and reducing task scheduling costs.

TAMPI [48] 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 safely use blocking MPI primitives inside tasks, without the risk of deadlock. Without TAMPI, a blocking MPI primitive blocks not only the task but also the underlying thread that runs it. Even if a normal MPI implementation avoids busy waiting, allowing the hardware thread to become idle, 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.

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 3 posts the non-blocking MPI_Irecv on line 6 to receive the contents of an array. It then calls TAMPI_Iwait, on line 7, 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 [47] to delay 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, it is not necessary to unblock and re-schedule the first task. TAMPI will use the external events API to release its dependencies, at which point the task on line 9 can begin 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 and OpenMP, asynchronous TAMPI and OpenMP/OmpSs-2, and OmpSs-2@Cluster with distributed taskiter. The benchmark is an in-place 2D stencil calculation where each element is updated based on the values of the elements above and to the left from the current timestep and the elements to the right and

```
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(in:matrix[y-1][x]) depend(in:matrix[y][x-1]) \
        depend(in:matrix[y][x+1]) depend(in:matrix[y+1][x]) \
        depend(inout:matrix[y][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(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]) \
        depend(inout:matrix[y][x]) \
        {
           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(in:matrix[y-1][x]) depend(in:matrix[y][x-1]) \
        depend(in:matrix[y][x+1]) depend(in:matrix[y+1][x]) \
        depend(inout:matrix[y][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

Figure 5: Gauss–Seidel 2D heat equation using three different programming models: fork–join MPI + OpenMP tasks, asynchronous TAMPI + OpenMP/OmpSs-2 tasks and OmpSs-2@Cluster with distributed taskiter. Each element is updated based on the values above and to the left from the current timestep and the values to the right and below from the previous timestep. The matrix is a grid of NBY×NBX blocks, each of size BSY×BSX. The MPI+OpenMP and TAMPI+OpenMP/OmpSs-2 variants are hard-coded to distribute the processes by rows, and the number of local rows of blocks is equal to NBY_LOCAL. 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.

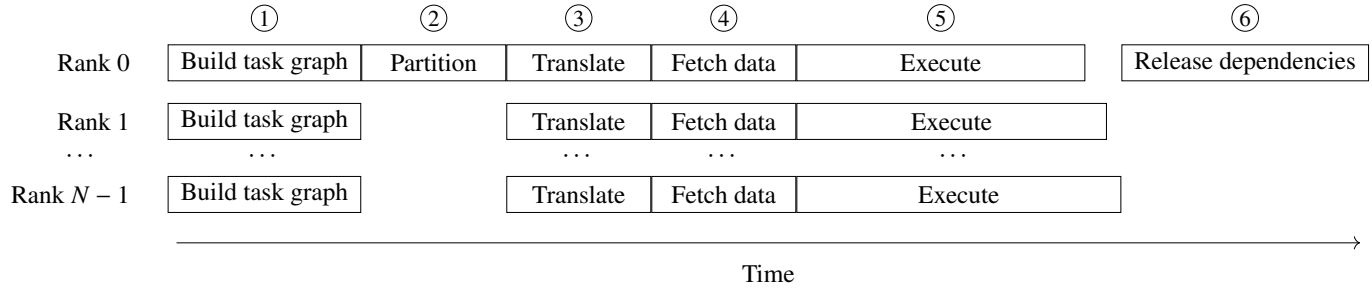


Figure 6: Illustration of the execution process for a distributed taskiter. This figure gives an overview of the sequence of steps and where they are executed, but it does not quantify the relative durations of the steps, which are not to scale.

below from the previous timestep. The entire matrix is a grid of 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 39 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 40.

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.

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. Firstly, 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. Secondly, the accesses to the blocks above and below the current block have been made more precise, since the task only reads a single row of these blocks rather than the whole block.[3] The MPI + OpenMP and TAMPI + OpenMP/OmpSs-2 versions correctly perform data transfers of a single row of each block, i.e, each of size BSX elements, since that is the only data accessed by the neighboring rank. Without this change to the distributed taskiter version, the oversized accesses on the tasks would mislead the runtime, forcing it to send whole blocks, each of size BSY×BSX elements. We employ the fragmented regions dependency system [43], which is enabled by default in OmpSs-2@Cluster, allowing correct enforcement of dependencies between tasks having accesses to full and partial blocks.

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 taskiter on SMP [6], 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 [3], 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 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

Compilation is done in exactly the same way as taskiter on SMP, as the programmer's model in Section 4 does not require

[3] It is unnecessary to increase the precision of the blocks to the left and to the right, due to the data distribution by rows among the nodes. But doing so is straightforward using multidependencies [54] and would not introduce any overheads during execution.

```
1  #pragma oss taskiter depend(weakinout:x,y,a,b)
2  for(int it=0; it<NUM_ITERATIONS; it++) {
3    // Task 1
4    #pragma oss task depend(in:x,y) depend(out:a) node(0)
5    { ··· }
6    // Task 2
7    #pragma oss task depend(in:a,y) depend(out:b) node(1)
8    { ··· }
9    // Task 3
10   #pragma oss task depend(in:a,b) depend(out:x) node(0)
11   { ··· }
12   // Task 4
13   #pragma oss task depend(in:x,b) depend(out:y) node(1)
14   { ··· }
15 }
```

Figure 7: Example OmpSs-2@Cluster distributed taskiter implementation of the program of Figure 3a, with the mapping from task to rank indicated using the "node" clause.

any changes to the compiler. The compiler encapsulates the loop body as a task, in a similar way to a taskloop or taskfor. The taskiter task is passed to the runtime together with the loop bounds and a flag to identify it as a taskiter.

5.2. Execute taskiter parent task to build full task graph

The taskiter becomes ready following the same condition as any other task, i.e. as soon as all of its strong accesses, if any, are satisfied. The execution of the taskiter begins at Step ① of Figure 6. The original node (typically Rank 0) creates and offloads a parentless copy of the taskiter to each other rank, and then it executes the taskiter itself. By running the taskiter task, each rank builds a local copy of the full task dependency graph for a single iteration. When the taskiter has the *unroll* clause, this "single iteration" may of course be more than one iteration of the underlying 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 [8] and OmpSs@cloudFPGA [24] (see Section 8). The current prototype uses a static partition controlled by the node clause.

Figure 7 updates the example program of Figure 3a 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 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 control the ordering and overlap of communication and computation. A send task has an *in* access on the data to send, since the MPI send only needs to read the latest version of the data. A receive task has an *out* access, since the MPI receive will update its buffer with the new version of the data.

The algorithm to add the send and receive tasks is shown in Figure 8. It starts from the top map, which is an existing data structure that maps each region to the first task that accesses it. 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. When using the regions dependency system, an extra pass 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. The tasks that were created locally by the taskiter parent in Step ①, but 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 any data transfers for the cyclic edges. 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 transfers 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 all 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, needs to create 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 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 algorithm in Figure 8 ensures that the iterations of each

```
1  mpiTag ← 0                                              ▷ Current MPI tag
2  initialValues ← ∅                ▷ Data that may need fetching for iteration 0
3  for all (region, access) ∈ topMapAccesses do
4      lastWriter ← none                          ▷ Last writer of region by sequential order
5      readerRanks ← ∅      ▷ Ranks with valid copy of latest version of region
6      initialReaderRanks ← ∅  ▷ Reading ranks of region from prev. iteration
7      ▷ Add send and receive tasks for non-cyclic dependencies ◁
8      while access ≠ none do
9          if access.type ≠ OUT and access.rank ∉ readerRanks then
10             ▷ Task reads data, but it is not yet present locally ◁
11             if lastWriter ≠ none then
12                 if currentRank = lastWriter.rank then
13                     Insert send task of region with tag mpiTag before access.task
14                 else if currentRank = access.rank then
15                     Insert receive task of region with tag mpiTag before
                       access.task
16                 mpiTag ← mpiTag + 1
17             readerRanks ← readerRanks ∪ access.rank
18         if access.type = OUT or access.type = INOUT then
19             ▷ Task writes data ◁
20             if lastWriter = none then
21                 initialReaderRanks ← readerRanks
22             lastWriter ← access.task
23             readerRanks ← {currentRank}
24         access ← access.next
25     ▷ Add send and receive tasks for cyclic dependencies ◁
26     for all initialReaderRank ∈ initialReaderRanks do
27         if initialReaderRank ∉ readerRanks then
28             if currentRank = lastWriter.rank then
29                 Insert send task of region with tag mpiTag at end
30             else if currentRank = initialReaderRank then
31                 Insert receive task of region with tag mpiTag at end
32                 initialValues ← initialValues ∪ region
33             mpiTag ← mpiTag + 1
34             readerRanks ← readerRanks ∪ initialReaderRank
```

Figure 8: Insertion of communication (send and receive) tasks. Communication is done by tasks, ensuring asynchronous execution with maximum communication–computation overlap. This deterministic algorithm runs on all ranks on the same full task graph, so sends and receives on different ranks will always match.

send and receive task are always serialized, irrespective of the algorithm used to execute the tasks (Section 5.5, which serializes the iterations of any particular task in any case). 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 process is the same as for SMP, and the result, for the example program, is shown in Figure 9b.

5.4.3. Fetch input data

As soon as Step ③ has finished on the current node, Step ④ fetches all of the input data that is needed by the taskiter. There is no need for a global barrier between Steps ③ and ④. The algorithm in Figure 8 has already determined, in *initialValues*, all the regions whose initial version, before the taskiter, is read by the first iteration of at least one task. These regions will require fetching to this node, unless the node already has a copy of the data. The runtime uses its normal data transfer mechanism, which checks whether a data transfer is actually needed, merges contiguous data transfers and programs the data transfers using non-blocking MPI calls.

5.5. Execute loop iterations

Once the data transfers in Step ④, 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.

The local graph is executed in the same way as taskiter on SMP. 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 [3].

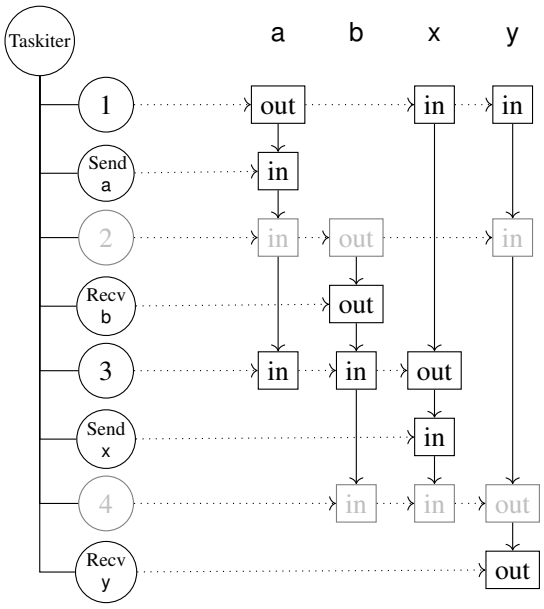
5.5.1. Non-constant number of iterations

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 the approach on SMP 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. Similarly to the SMP implementation, 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 the same way as SMP, in order to support overlapping of tasks from different iterations.

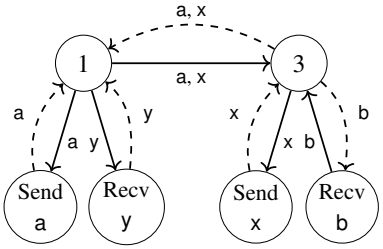
5.5.2. Release accesses

Once all local tasks on a remote (non-Rank 0) rank have completed all iterations, then a Task Finished message is sent to Rank 0 in the normal way. Once Rank 0 has completed its own

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



(a) Local dependency graph on Rank 0 after inserting the send and receive tasks. The tasks not executed on Rank 0, i.e., Task 2 and Task 4, are disabled and colored in gray.



(b) Local DCTG on Rank 0.

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

iterations and received notification from all other nodes, it completes the taskiter and releases its accesses. Since Rank 0 knows the partitioning of tasks across ranks (as does every rank), the data locations in the dependency system are updated to correspond to the rank that executes the last writer. If there is no last writer, because the data is only read, then the original location is not modified.

6. Evaluation methodology

6.1. Hardware and software platform

The experiments in this paper were performed on up to 32 nodes of the general-purpose block of the MareNostrum 4 supercomputer [9]. MareNostrum 4 comprises 3456 compute nodes, each with two 24-core Intel Xeon Platinum sockets. We use normal memory capacity nodes, which have 96 GB physical memory (2 GB per core). The interconnect is 100 Gb/s Intel Omni-Path with a fat tree. GCC 7.2.0 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 2018.4, and the benchmarks use the BLAS functions provided by Intel MKL 2018.4. 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 in configurations of 2 processes per node (one per NUMA node), following previous work [3], which found that using one process per socket led to better and less variable results, due to the more effective use of NUMA locality. Results are given for 1 to 32 nodes, for a total of up to 64 MPI processes.

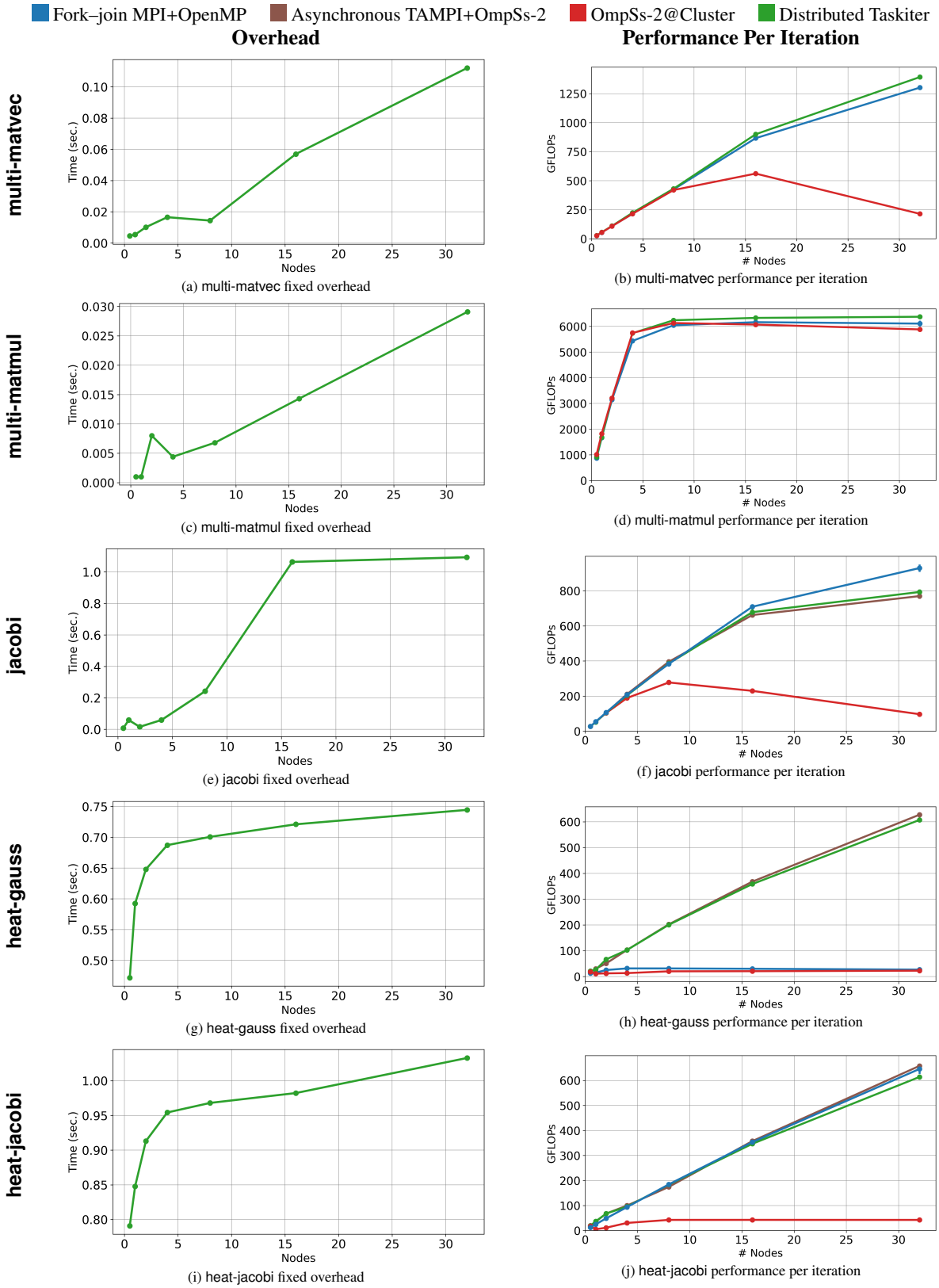
multi-matvec is a sequence of identical dense double-precision matrix–vector multiplications, with the matrix distributed by rows and without dependencies between iterations.[5]. It has fine-grained tasks with complexity $O(n^2)$ and no inter-node data transfers. multi-matmul is a sequence of dense double-precision 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 scalability

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 fixed 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 (GFLOPS/s on the y-axis). The colors distinguish the four programming models: blue for fork–join MPI + OpenMP, brown for TAMPI + OmpSs-2, red for the original OmpSs-2@Cluster implementation, and green for the distributed taskiter. 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, but in almost all cases the error bars are too small to see.

[5] Prior work [3] referred to these same benchmarks as matvec and matmul



(a) multi-matvec fixed overhead

(b) multi-matvec performance per iteration

(c) multi-matmul fixed overhead

(d) multi-matmul performance per iteration

(e) jacobi fixed overhead

(f) jacobi performance per iteration

(g) heat-gauss fixed overhead

(h) heat-gauss performance per iteration

(i) heat-jacobi fixed overhead

(j) heat-jacobi performance per iteration

Figure 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 fixed overhead and the right-hand column gives the performance per iteration, excluding the fixed overhead. We see that the maximum overhead of distributed taskiter is just 1.1 seconds (jacobi on 32 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.

Benchmark	Description	Characteristics	Parameters
multi-matvec	Repeated dense matrix–vector multiplication	Small tasks and no communication	Matrix size: 32,768×32,768 elements Number of iterations: 500
multi-matmul	Repeated dense matrix–matrix multiplication	Large tasks and no communication	Matrix size: 32,768×32,768 elements Number of iterations: 6
jacobi	Jacobi iteration [3]	All-to-all communication, good fit for fork–join parallelism	Matrix size: 32,768×32,768 elements Number of iterations: 400
heat-gauss	2D stencil computation with Gauss–Seidel updates	Nearest-neighbor communication, wavefront parallelism suited to asynchronous tasks	Matrix size: 32,768×32,768 elements Number of iterations: 100
heat-jacobi	2D stencil computation with Jacobi updates	Nearest-neighbor communication, good fit for fork–join parallelism	Matrix size: 32,768×32,768 elements Number of iterations: 100

Table 1: Evaluation benchmarks

Figure 10a shows the overhead for multi-matvec, which has a maximum value of just 0.11 seconds on 32 nodes. This overhead corresponds to Steps ① to ④ and Step ⑥ in Section 5, and for this benchmark it scales roughly linearly with the number of nodes, since the optimal block size corresponds to a small number of tasks per core, and all tasks always have the same number of accesses. Looking at Figure 10b, we see that, after paying this small cost, the distributed taskiter variant achieves similar scaling behaviour to the baseline fork–join MPI version. There is a slight improvement over fork–join MPI by 7.0% on 32 nodes, likely because of minor differences between the OpenMP and Nanos6@Cluster runtimes. This result is a large improvement compared with the original OmpSs-2@Cluster implementation, which scales to just 8 nodes and is 6 times slower than fork–join MPI on 32 nodes. This benchmark has no inter-node communication, so the poor scaling of the OmpSs-2@Cluster implementation is due to the control message overhead for task offloading and dependency management, which is entirely eliminated by the distributed taskiter approach. Since there is no inter-node communication, the asynchronous TAMPI + OmpSs-2 results have been omitted.

Figure 10c shows the fixed overhead for multi-matmul, which has a maximum value of just 0.029 seconds on 32 nodes. Again, the overhead scales roughly linearly with the number of nodes. As seen in Figure 10d, all three variants scale similarly for this benchmark. The drop in scalability beyond 4 nodes is due to unusual behavior from the MKL library, which achieves approximately 3× higher throughput for block sizes of 256 elements or more. For a 32768×32768 matrix, the optimal block size does not fully use the available compute resources when there are 8 or more nodes.

Figure 10e shows that the fixed overhead for jacobi grows roughly quadratically from 1 to 16 nodes. This is due to the benchmark's all-to-all communication, which means that the number of tasks and the number of accesses per task both grow roughly linearly in the number of cores, which combine to cause the quadratic growth. Beyond 16 nodes, it is no longer beneficial to subdivide the tasks. The maximum overhead on 32 nodes is 1.1 seconds. In Figure 10f, we see that the distributed taskiter approach matches the fork–join MPI version, within 4.4% on up to 16 nodes, but it drops to 15% below the fork–join MPI version on 32 nodes. There are two reasons for this. Firstly, the fork–join MPI version uses collective communication, whereas distributed taskiter uses point-to-point communication. Secondly, the distributed taskiter has asynchronous communication inside tasks instead of fork–join parallelism. We see that the results match closely, within 3%, those for the asynchronous TAMPI + OmpSs-2 version on up to 32 nodes, which also has point-to-point communication inside tasks. Future work may investigate ways to use collective communication or merge communication into existing tasks. In any case, the results already greatly outperform the original OmpSs-2@Cluster, which is 9.6 times slower than fork–join MPI + OpenMP.

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 0.47 seconds on 1 node to 0.74 seconds on 32 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 11.4 times faster than fork–join MPI + OpenMP on 32 nodes. The distributed taskiter version achieves similar performance, at 11.0 times faster than fork–join MPI + OpenMP on 32 nodes. In contrast, the OmpSs-2@Cluster implementation has even worse performance than fork–join MPI + OpenMP, being 2.4 times *slower*, on 32 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 32 nodes. The overhead is almost constant, rising to a maximum of just over 1.0 seconds on 32 nodes. Finally, in Figure 10j, all versions except the original OmpSs-2@Cluster implementation achieve similar scaling to at least 32 nodes. The distributed taskiter version is within 5.0% of the performance per iteration of fork–join MPI, which is a dramatic improvement in comparison with the original OmpSs-2@Cluster implementation which is 15 times slower than fork–join MPI.

Overall, these results show that the fixed 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 24 threads in each process. After paying that small cost, the

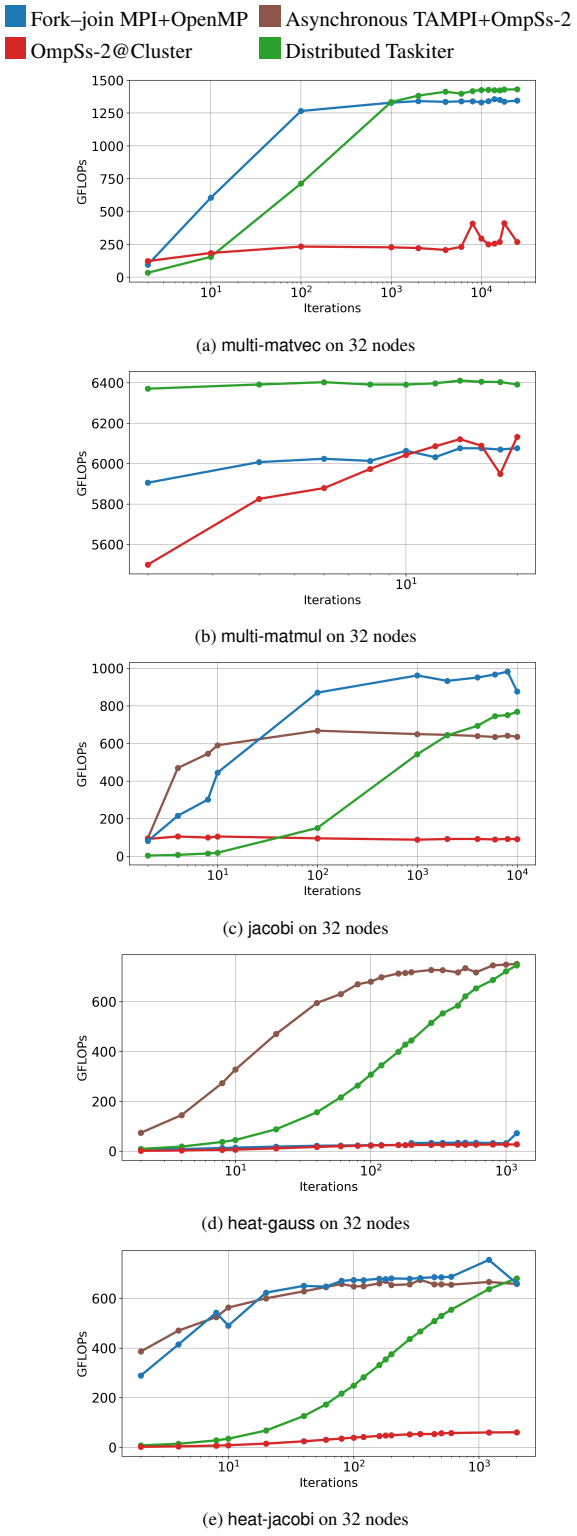
performance-per-iteration matches or exceeds fork–join MPI + OpenMP, with a slight drop of 15% only for the jacobi benchmark on 32 nodes. For heat-gauss, which benefits from asynchronous communication, the performance is similar to asynchronous TAMPI + OmpSs-2, at 11.0 times faster than fork–join MPI + 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 15 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, in all cases executing on 32 nodes (64 processes). The *x*-axis is the number of iterations, on a log scale. The *y*-axis is the overall performance, in GFLOPS/s, which, unlike the right-hand column of Figure 10, includes the fixed 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 implementation achieves higher performance for multi-matmul, heat-gauss and heat-jacobi than the original OmpSs-2@Cluster version from the first iteration. For multi-matvec, it exceeds the original version from 10–100 and 100-1000 iterations, respectively. The distributed taskiter version has slightly higher performance than fork–join MPI + OpenMP, from 100 or more iterations for multi-matvec and from the first iteration for multi-matmul (note the zoomed *y*-axis for multi-matmul, in order to expose the small differences between versions). For jacobi, distributed taskiter's performance steadily increases up to about 15% below that of fork–join MPI + OpenMP. For heat-gauss, the two versions that allow asynchronous overlap of timesteps, i.e., the TAMPI + OmpSs-2 and distributed taskiter versions, 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 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 24 threads in each process.



(a) multi-matvec on 32 nodes

(b) multi-matmul on 32 nodes

(c) jacobi on 32 nodes

(d) heat-gauss on 32 nodes

(e) heat-jacobi on 32 nodes

Figure 11: Overall performance, including all overheads, as a function of the number of iterations. For multi-matmul, heat-gauss and heat-jacobi, the distributed taskiter version outperforms the original OmpSs-2@Cluster version from the first iteration (unrolled by two). In many cases, few iterations are required to achieve performance close to fork–join MPI + OpenMP.

8. Related work

8.1. MPI, PGAS and hybrid approaches

Message Passing Interface (MPI) [37] 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 [38, 53, 33] and libraries [28, 20] 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 [40], OpenACC [39], CUDA [35], or similar, have been under study for at least twenty years [45, 30]. Many applications use a fork–join approach, where processes alternate between sequential communication and parallel computation phases, which hinders inter- and intra-node parallelism. Habanero-C MPI (HCMPI) [22] automatically supports fine-grained overlapping of communication and computation, as it converts each MPI call into an asynchronous task. TAMPI [48] (see Section 2.4) is a more flexible approach that allows tasks to safely and efficiently call MPI primitives. All MPI+X approaches suffer from the fundamental issues of MPI, i.e., the programmer has to handle data distribution, synchronization and load balancing, as well as inserting message sends and receives. They also require the programmer to split the parallelism between shared and distributed memory models.

DASH [27] 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.

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 Graph (STG) formulation. In the latter case, the STG 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 STG 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** [17, 29] is a distributed task-based model designed for scalability on distributed heterogeneous architectures. It is the basis for the DPLASMA library for dense linear algebra, which was the original motivation for PaRSEC. Its original formulation builds a parameterized Directed Acyclic Graph (DAG) [23], which describes the dependencies between tasks in an algebraic way in terms of the iteration variables. This model is very complex to develop programs, and it is not as expressive as other task models, being tailored for affine loops that are amenable with polyhedral analysis [19]. Our approach also avoids unrolling the whole dependency graph, but it is simpler and specialized for iterative applications, and it requires just one pragma to identify such a loop.

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 [8] extends StarPU [5] to support distributed memory tasking using MPI. In this framework, top-level tasks are mapped to nodes using an owner-computes model. **YarKhan**'s [55] extension of QUARK uses a deterministic mapping of task to rank based on data distribution, and it also uses MPI for communication. **TBLAS** [52] takes a similar approach to target clusters of CPUs, each with multiple GPUs. **OmpSs@cloudFPGA** [24] 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** [17, 29] 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. **OMPC** [56] 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. **OmpSs-1@Cluster** [18] 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 32 nodes.

Legion [16] 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 [50]. 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++ [41] 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** [31] 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** [21] is an object-oriented programming language for high-productivity programming that spawns asynchronous computations, with the programmer responsible for PGAS data distribution.

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** [34] is a Java, C/C++ and Python framework to run parallel applications on clusters, clouds and containerized platforms. It is a sequential task-based model similar to OmpSs, but dependencies are tracked through files or objects rather than the program's virtual address space. **Pegasus** [25] is another workflow management system that uses a DAG of tasks and dependencies. **GPI-Space** [46] is a fault-tolerant execution platform for data-intensive applications. It supports coarse-grained tasks that helps decouple 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 remaining interoperable with the OmpSs-2@Cluster model. While the existing OmpSs-2@Cluster implementation scales to only about 4 or 8 nodes with medium-scale tasks, our approach scales to at least 32 nodes, with a maximum slowdown of 15%, 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 11.0 times higher performance on 32 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, 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). 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/10.13039/501100011-033 and by "ESF Investing in your future", as well as by the *Generalitat de Catalunya* (2017-SGR-1414).

References

[1] "Clang: a C language family frontend for LLVM," Jan 2024, accessed: 2024-01-18. [Online]. Available: https://clang.llvm.org/

[2] "The LLVM compiler infrastructure," Jan 2024, accessed: 2024-01-18. [Online]. Available: https://llvm.org/

[3] J. Aguilar Mena, O. Shaaban, V. Beltran, P. Carpenter, E. Ayguadé, and J. Labarta, "OmpSs-2@Cluster: Distributed memory execution of nested OpenMP-style tasks," in *European Conference on Parallel Processing: Euro-Par*, 2022. [Online]. Available: https://doi.org/10.1007/978-3-031-12597-3_20

[4] J. Aguilar Mena, O. Shaaban, V. Lopez, M. Garcia, P. Carpenter, E. Ayguadé, and J. Labarta, "Transparent load balancing of MPI programs using OmpSs-2@Cluster and DLB," in *51st International Conference on Parallel Processing (ICPP)*, 2022. [Online]. Available: https://doi.org/10.1145/3545008.3545045

[5] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. P. Thibault, "Achieving high performance on supercomputers with a sequential task-based programming model," *IEEE Transactions on Parallel and Distributed Systems*, 2017. [Online]. Available: https://doi.org/10.1109/TPDS.2017.2766064

[6] D. Álvarez and V. Beltran, "Optimizing iterative data-flow scientific applications using directed cyclic graphs," *IEEE access*, 2023. [Online]. Available: https://doi.org/10.1109/ACCESS.2023.3269902

[7] D. Álvarez, K. Sala, M. Maroñas, A. Roca, and V. Beltran, "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: Association for Computing Machinery, 2021, p. 334–347. [Online]. Available: https://doi.org/10.1145/3437801.3441601

[8] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault, "StarPU-MPI: Task programming over clusters of machines enhanced with accelerators," in *European MPI Users' Group Meeting*. Springer Berlin Heidelberg, 2012, pp. 298–299. [Online]. Available: https://doi.org/10.1007/978-3-642-33518-1_40

[9] Barcelona Supercomputing Center, "MareNostrum 4 (2017) System Architecture," 2017. [Online]. Available: https://www.bsc.es/marenostrum/marenostrum/technical-information

[10] ——. (2021) Influence in OpenMP - OmpSs-2 specification. [Online]. Available: https://pm.bsc.es/ftp/ompss-2/doc/spec/introduction/openmp.html

[11] ——. (2021) Nanos6. [Online]. Available: https://github.com/bsc-pm/nanos6

[12] ——. (2021) OmpSs-2 specification. [Online]. Available: https://pm.bsc.es/ftp/ompss-2/doc/spec/

[13] ——. (2021) Taskiter. [Online]. Available: https://github.com/bsc-pm/ompss-2-cluster-releases#taskiter

[14] ——, "OmpSs-2@Cluster releases," 2022. [Online]. Available: https://github.com/bsc-pm/ompss-2-cluster-releases

[15] ——. (2024) OmpSs-2 LLVM compiler infrastructure. [Online]. Available: https://github.com/bsc-pm/llvm

[16] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11. [Online]. Available: https://doi.org/10.1109/SC.2012.71

[17] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Luszczek, and J. Dongarra, "Dense linear algebra on distributed heterogeneous hardware with a symbolic DAG approach," Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), Tech. Rep., 2012. [Online]. Available: https://www.osti.gov/servlets/purl/1173290

[18] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta, "Productive programming of GPU clusters with OmpSs," in *IEEE 26th International Parallel and Distributed Processing Symposium*, 5 2012. [Online]. Available: https:doi.org//10.1109/IPDPS.2012.58

[19] P. Cardosi and B. Bramas, "Specx: a c++ task-based runtime system for heterogeneous distributed architectures," *arXiv preprint arXiv:2308.15964*, 2023.

[20] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing OpenSHMEM: SHMEM for the PGAS community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, 2010, pp. 1–3. [Online]. Available: https://doi.org/10.1145/2020373.2020375

[21] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "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*, ser. OOPSLA '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 519–538. [Online]. Available: https://doi.org/10.1145/1094811.1094852

[22] S. Chatterjee, S. Tasırlar, Z. Budimlic, V. Cavé, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with MPI," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 712–725. [Online]. Available: https://doi.org/10.1109/IPDPS.2013.78

[23] M. Cosnard and M. Loi, "Automatic task graph generation techniques," in *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, vol. 2, 1995, pp. 113–122 vol.2.

[24] J. M. de Haro, R. Cano, C. Álvarez, D. Jiménez-González, X. Martorell, E. Ayguadé, J. Labarta, F. Abel, B. Ringlein, and B. Weiss, "OmpSs@cloudFPGA: An FPGA task-based programming model with message passing," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 828–838.

[25] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. Berriman, J. Good, A. Laity, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, Jan 2005. [Online]. Available: https://doi.org/10.1155/2005/128026

[26] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: a proposal for programming heterogeneous multi-core architectures," *Parallel processing letters*, vol. 21, no. 02, pp. 173–193, 2011. [Online]. Available: https://doi.org/10.1142/S0129626411000151

[27] K. Fürlinger, J. Gracia, A. Knüpfer, T. Fuchs, D. Hünich, P. Jungblut, R. Kowalewski, and J. Schuchart, "DASH: Distributed data structures and parallel algorithms in a global address space," in *Software for Exascale Computing-SPPEXA 2016-2019*. Springer International Publishing, 07 2020, pp. 103–142. [Online]. Available: https://doi.org/10.1007/978-3-030-47956-5_6

[28] D. Grünewald and C. Simmendinger, "The GASPI API specification and its implementation GPI 2.0," in *7th International Conference on PGAS Programming Models*, vol. 243, 2013, p. 52.

[29] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "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*, 11 2017, pp. 1–8. [Online]. Available: https://doi.org/10.1145/3148226.3148233

[30] G. Jost, H.-Q. Jin, F. F. Hatay *et al.*, "Comparing the OpenMP, MPI, and hybrid programming paradigm on an SMP cluster," in *European Workshop on OpenMP and Applications 2003*, 2003. [Online]. Available: https://ntrs.nasa.gov/citations/20030107321

[31] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task based programming model in a global address space," in *8th International Conference on Partitioned Global Address Space Programming Models*, 2014. [Online]. Available: https://doi.org/10.13140/2.1.2635.5204

[32] J. Klinkenberg, P. Samfass, M. Bader, C. Terboven, and M. Müller, "CHAMELEON: Reactive load balancing for hybrid MPI+OpenMP task-parallel applications," *Journal of Parallel and Distributed Computing*, vol. 138, 12 2019. [Online]. Available: https://doi.org/10.1016/j.jpdc.2019.12.005

[33] J. Lee, M. T. Tran, T. Odajima, T. Boku, and M. Sato, "An extension of XcalableMP PGAS lanaguage for multi-node GPU clusters," in *Euro-Par 2011: Parallel Processing Workshops: CCPI, CGWS, HeteroPar, HiBB, HPCVirt, HPPC, HPSS, MDGS, ProPer, Resilience, UCHPC, VHPC, Bordeaux, France, August 29–September 2, 2011, Revised Selected Papers, Part I 17*. Springer, 2012, pp. 429–439. [Online]. Available: https://doi.org/10.1007/978-3-642-29737-3_48

[34] F. Lordan, E. Tejedor, J. Ejarque, R. Rafanell, J. Alvarez, F. Marozzo, D. Lezzi, R. Sirvent, D. Talia, and R. M. Badia, "ServiceSs: An interoperable programming framework for the cloud," *Journal of grid computing*, vol. 12, no. 1, 2014. [Online]. Available: https://doi.org/10.1007/s10723-013-9272-5

[35] D. Luebke, "CUDA: Scalable parallel programming for high-performance scientific computing," in *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*. IEEE, 2008, pp. 836–838. [Online]. Available: https://doi.org/10.1109/ISBI.2008.4541126

[36] J. A. Mena, "Methodology for malleable applications on distributed memory systems," Ph.D. dissertation, Universitat Politècnica de Catalunya, 2022. [Online]. Available: http://dx.doi.org/10.5821/dissertation-2117-380814

[37] MPI Forum, "MPI documents." [Online]. Available: https://www.mpi-forum.org/docs/

[38] R. W. Numrich and J. Reid, "Co-array Fortran for parallel programming," in *ACM SIGPLAN Fortran Forum*, vol. 17, no. 2. ACM New York, NY, USA, 1998, pp. 1–31. [Online]. Available: https://doi.org/10.1145/289918.289920

[39] OpenACC Organization, "OpenACC: Directives for accelerators," 2011. [Online]. Available: http://www.openacc-standard.org

[40] OpenMP Architecture Review Board, "OpenMP Application Programming Interface, Version 5.2," 11 2021, accessed: 2022-04-19. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[41] Parallel Programming Lab, Dept of Computer Science, University of Illinois. (2023) Charm++ documentation. [Online]. Available: https://charm.readthedocs.io/en/latest/index.html

[42] J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé, "Improving the integration of task nesting and dependencies in OpenMP," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 809–818. [Online]. Available: https://doi.org/10.1109/IPDPS.2017.69

[43] J. M. Perez, R. M. Badia, and J. Labarta, "Handling task dependencies under strided and aliased references," in *Proceedings of the 24th ACM International Conference on Supercomputing*, 2010, pp. 263–274. [Online]. Available: https://doi.org/10.1145/1810085.1810122

[44] J. Pérez, R. M. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," 2008, pp. 142–151. [Online]. Available: https://doi.org/10.1109/CLUSTR.2008.4663765

[45] R. Rabenseifner and G. Wellein, "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, 2005, pp. 409–425. [Online]. Available: https://doi.org/10.1007/3-540-27170-8_31

[46] T. Rotaru, M. Rahn, and F.-J. Pfreundt, "MapReduce in GPI-Space," in *Euro-Par 2013: Parallel Processing Workshops*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 43–52. [Online]. Available: https://doi.org/10.1007/978-3-642-54420-0_5

[47] K. Sala, S. Macià, and V. Beltran, "Combining one-sided communications with task-based programming models," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 528–541. [Online]. Available: https://doi.org/10.1109/Cluster48925.2021.00024

[48] K. Sala, X. Teruel, J. M. Perez, A. J. Peña, V. Beltran, and J. Labarta, "Integrating blocking and non-blocking MPI primitives with task-based programming models," *Parallel Computing*, vol. 85, pp. 153–166, 2019. [Online]. Available: https://doi.org/10.1016/j.parco.2018.12.008

[49] O. Shaaban, J. Aguilar, V. Beltran, P. Carpenter, E. Ayguadé, and J. L. Mancho, "Automatic aggregation of subtask accesses for nested OpenMP-style tasks," in *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2022, pp. 315–325. [Online]. Available: https://doi.org/10.1109/SBAC-PAD55451.2022.00042

[50] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "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*, 2015, pp. 1–12. [Online]. Available: https://doi.org/10.1145/2807591.2807629

[51] L. Smith and M. Bull, "Development of mixed mode MPI / OpenMP applications," *Sci. Program.*, vol. 9, no. 2,3, p. 83–98, aug 2001.

[52] F. Song and J. Dongarra, "A scalable framework for heterogeneous GPU-based clusters," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, 2012, pp. 91–100.

[53] UPC Consortium, "Berkeley UPC – Unified Parallel C," 2024. [Online]. Available: https://upc.lbl.gov/

[54] R. Vidal, M. Casas, M. Moretó, D. Chasapis, R. Ferrer, X. Martorell, E. Ayguadé, J. Labarta, and M. Valero, "Evaluating the impact of OpenMP 4.0 extensions on relevant parallel workloads," in *OpenMP: Heterogenous Execution and Data Movements: 11th International Workshop on OpenMP, IWOMP 2015, Aachen, Germany, October 1-2, 2015, Proceedings 11*. Springer, 2015, pp. 60–72. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-24595-9_5

[55] A. YarKhan, "Dynamic task execution on shared and distributed memory architectures," Ph.D. dissertation, University of Tennessee, 2012. [Online]. Available: https://trace.tennessee.edu/cgi/viewcontent.cgi?article=2774&context=utk_graddiss

[56] H. Yviquel, M. Pereira, E. Francesquini, G. Valarini, G. Leite, P. Rosso, R. Ceccato, C. Cusihualpa, V. Dias, S. Rigo, A. Souza, and G. Araujo, "The OpenMP Cluster programming model," in *Workshop Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP Workshops '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3547276.3548444