OmpSs-2@Cluster: Distributed memory execution of nested OpenMP-style tasks

Jimmy Aguilar Mena[1], Omar Shaaban[1], Vicenç Beltran[1], Paul Carpenter[1], Eduard Ayguade[1], and Jesus Labarta Mancho[1]

Barcelona Supercomputing Center

Abstract. State-of-the-art programming approaches generally have a strict division between intra-node shared memory parallelism and inter-node MPI communication. Tasking with dependencies offers a clean, dependable abstraction for a wide range of hardware and situations within a node, but research on task offloading between nodes is still relatively immature. This paper presents a flexible task offloading extension of the OmpSs-2 programming model, which inherits task ordering from a sequential version of the code and uses a common address space to avoid address translation and simplify the use of data structures with pointers. It uses weak dependencies to enable work to be created concurrently. The program is executed in distributed dataflow fashion, and the runtime system overlaps the construction of the distributed dependency graph, enforces dependencies, transfers data, and schedules tasks for execution. Asynchronous task parallelism avoids synchronization that is often required in MPI+OpenMP tasks. Task scheduling is flexible, and data location is tracked through the dependencies. We wish to enable future work in resiliency, scalability, load balancing and malleability, and therefore release all source code and examples open source.

1 Introduction

The dominant programming approach for scientific and industrial computing on clusters is MPI+X. While there are a variety of approaches within the node, described by the "X", such as OpenMP, OmpSs, OpenACC and others, the *de facto* standard for programming multiple nodes is MPI. In all cases the program must combine two fundamentally different programming models, which is difficult to get right [30,29]. The tasking approach of OpenMP and OmpSs offers an open, clean and stable way to improve hardware utilization through asynchronous execution while targeting a wide range of hardware, from SMPs, to GPUs, to FPGAs. This paper extends the same approach, of OmpSs-2 tasking, to multiple nodes. We develop OmpSs-2@Cluster, which provides a simple path to move an OmpSs-2 program from single node to small- to medium-scale clusters. We also present the runtime techniques that allow overlapping of the construction of the distributed dependency graph, efficient concurrent enforcing of dependencies, data transfers among nodes, and task execution.

A number of research groups are looking into tasks as a model for all scales from single threads and accelerators to clusters of nodes, as outlined in Section 7.

Our approach is unique, in that, in many cases, a functional multi-node version of an existing OmpSs-2 program can be obtained simply by changing the configuration file supplied to the runtime system. The meaning of the program are defined by the sequential semantics of the original program, which simplifies development and maintenance. All processes use the same virtual memory layout, which avoids address translation and allows direct use of existing data structures with pointers. Improvements beyond the first version can be made incrementally, based on observations from performance analysis. Some optimizations that are well-proven within a single node, such as task nesting to overlap task creation and execution, [26] are a particular emphasis of OmpSs-2@Cluster, since they clearly have a greater impact when running across multiple nodes, due to the greater node-to-node latency and larger total number of execution cores.

The program is executed in distribution dataflow fashion, which is naturally asynchronous, with no risk of deadlock due to user error. In contrast, MPI+X programs often use a fork–join model, due to the difficulty in overlapping computation and communication [30]. We show how well-balanced applications have similar performance to MPI+OpenMP on up to 16 nodes. For irregular and unbalanced applications like Cholesky factorization, we get a 2× performance improvement on 16 nodes, compared with a high performance implementation using MPI+OpenMP tasks. All source code and examples are released open source [9].

2 Background

OmpSs-2 [8,7,9] is the second generation of the OmpSs programming model. It is open source and mainly used as a research platform to explore and demonstrate ideas that may be proposed for standardization in OpenMP. The OpenMP concept of data dependencies among tasks was first proven in OmpSs. Like OpenMP, OmpSs-2 is based on directives that annotate a sequential program, and it enables parallelism in a dataflow way [27]. The model targets multi-cores and GPU/FPGA accelerators. This decomposition into tasks and data accesses is used by the source-to-source Mercurium [5] compiler to generate calls to the Nanos6 [6] runtime API. The runtime computes task dependencies and schedules and executes tasks, respecting the implied task dependency constraints and performing data transfers and synchronizations.

OmpSs-2 differs from OpenMP in the thread-pool execution model, targeting of heterogeneous architectures through native kernels, and asynchronous parallelism as the main mechanism to express concurrency. Task dependencies may be discrete (defined by start address), or regions with fragmentation [26].

OmpSs-2 extends the tasking model of OmpSs and OpenMP to improve task nesting and fine-grained dependences across nesting levels [26,2]. The `depend` clause is extended with `weakin`, `weakout` and `weakinout` dependency types, which serve as a linking point between the dependency domains at different nesting levels, without delaying task execution. They indicate that the task does not itself access the data, but its nested subtasks may do so. Any subtask that directly accesses data needs to include it in a `depend` clause in the non-weak

Feature	Description
Sequential semantics	Simplifies development, porting, and maintenance. Tasks can be defined at any nesting level and can be offloaded to any node.
Common address space	Simplifies porting of applications with complex data structures by supporting pointers and avoiding address translation.
Distributed dataflow execution	Task ordering and overlapping of data transfers with computation are automated, reducing synchronizations and avoiding risk of deadlock.
Distributed memory allocation	Informs runtime that memory is only needed by subtasks, reducing synchronization and data transfers. Provides data distribution affinity hint.
Minimizing of data transfers	The `taskwait on` and `taskwait noflush` directives help minimize unnecessary data transfers.
Early, late or auto release of dependencies	A tradeoff between parallelism and overhead is exposed through control over the release of dependencies.
Cluster query API and scheduling hint	Optional ability to instruct the runtime to control detailed behavior and optimize decisions.

Table 1: Key features of OmpSs-2@Cluster

variant. Any task that delegates accesses to a subtask must include the data in its `depend` clause in at least the weak variant. This approach enables effective parallelization of applications using a top-down methodology. The addition of weak dependences exposes more parallelism, allows better scheduling decisions and enables parallel instantation of tasks with dependencies between them.

3 OmpSs-2@Cluster programming model

The main features of OmpSs-2@Cluster are summarized in Table 1. Like OmpSs-2 on an SMP, tasks are defined by annotations to a program with *sequential semantics*, and offloadable tasks can be nested and defined at any nesting level. There is a *common address space* across cluster nodes, with data mapped to the same virtual address space on all nodes. As long as the task's accesses are described by dependencies, any data allocated on any node can be accessed at the same location on any other node. There is sufficient virtual address space on all modern 64-bit processors to support up to 65k cores with a typical 2 GB/core footprint. Almost any OmpSs-2 program can therefore be executed using OmpSs-2@Cluster and, conversely, if new features are ignored or implemented with a stub, any OmpSs-2@Cluster program is a valid OmpSs-2 program. This property minimizes porting effort and allows re-use of existing benchmarks.

Figure 1 shows an optimized matrix–matrix multiplication kernel using OmpSs-2@Cluster. Execution starts on node 0, which runs `main` as the first task. The example offloads one task per node then subdivides the work among the cores using a `task for`. The outer task with weak dependencies is an optimization to allow subtask creation to be overlapped with task execution, as shown in Section 6 (results). In general, the program as a whole is executed in a *distributed dataflow* fashion, with data transfers and data consistency managed by the runtime system. Data location is passed through the distributed dependency graph.

Compared with OmpSs-2 on SMP, OmpSs-2@Cluster has one new requirement for correctness (full dependency specification) and a few programming

```
void matmul(double *A, const double *B, const double *C, int dim, int ts)
{
  int rowsPerNode = dim / nanos6_get_num_cluster_nodes();

  for(int i = 0; i < dim; i += rowsPerNode) {
    #pragma oss task label("weakmatvec") \
      weakin(A[i*dim; rowsPerNode*dim]) weakin(B[0; dim*dim]) \
      weakout(C[i*dim; rowsPerNode*dim])
    {
      #pragma oss task for label("taskformatvec") \
        in(A[i*dim; rowsPerNode*dim]) in(B[0; dim*dim]) \
        out(C[i*dim; rowsPerNode*dim])
      for(int j = i; j < i + rowsPerNode; j += ts) {
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                    ts, dim, dim, 1.0, &A[j * dim], dim,
                    B, dim, 0.0, &C[j * dim], dim);
      }
    }
  }
}
```

Fig. 1: OmpSs-2@Cluster program: optimized dense matrix–matrix multiply, using a weak parent task to overlap task creation and execution.

model extensions to improve performance. Only a minor revision to the compiler is required to support these new features.

Full dependency specification: offloadable tasks (see below) require a full dependency specification, i.e., `in`, `out`, and `inout` dependencies must specify all accesses, rather than just the constraints needed for task ordering. This is the only reason that a valid OmpSs-2 program may not be a valid OmpSs-2@Cluster program, as in SMP systems accesses that are not needed to resolve dependencies may be omitted. This is not a new issue because a similar requirement exists for accelerators with separate memory spaces.

Distributed memory allocation: The new distributed malloc, `nanos6_dmalloc`, is an alternative memory allocation primitive for large data structures manipulated on multiple nodes. This call expresses three important distinguishing characteristics. Firstly, since the data is intended to be manipulated by concurrent tasks on several nodes, it can be assumed that the data is not used by the enclosing task, only its subtasks or descendants, similarly to a weak dependency. Secondly, since large allocations are infrequent and use significant virtual memory, it is efficient to centralize the memory allocator, as the overhead is tolerable and it leads to more efficient use of the virtual memory. Thirdly, it is a convenient place to provide a data distribution hint. The data distribution hint is communicated to all nodes, and is intended to help the scheduler improve load balance and data locality, using information from the programmer, if available. The hint does not mandate a particular data distribution, only the data affinity. The scheduler can take account of both the affinity and current location, depending on the chosen scheduling policy.

Minimising of data transfers: OmpSs-2@Cluster adds the `noflush` clause for taskwaits, in order to separate synchronization from data dependency. The contents of the memory allocated by `nanos6_dmalloc` and weak dependencies are by default noflush, so that tasks can wait for their child tasks without copying data that is not needed. The `noflush` variant is also useful for timing parts of the execution. When only a subset of locally-allocated data is needed by the enclosing task, the dependency and data transfer can be expressed using `taskwait on`.

Early, late or auto release of dependencies: As per the OmpSs-2 specification on SMPs, non-offloaded OmpSs-2@Cluster tasks by default early release all of their dependencies, so that data is passed directly to the successor task without additional synchronization. Late release of dependencies is possible, using the OmpSs-2 `wait` clause, which adds an implicit `taskwait` after the completion of the task body (and release of the stack). On OmpSs-2@Cluster, offloaded tasks by default have auto release of dependencies, which means that early release is supported to successors on the same node, but all other dependencies wait. The alternatives are available through the `wait` and `nowait` clauses.[1]

Cluster query API: There is also a simple API to read information about the execution environment: `nanos6_get_num_cluster_nodes()` returns the number of processes, and `nanos6_get_cluster_node_id()` returns the current rank.

Scheduling hint: The runtime schedules tasks, among and within nodes, taking account of current data location and/or affinity from the data distribution hint of `nanos6_dmalloc`. The programmer can override scheduling using the `node` clause on the `task` directive, which can specify the process that will execute the task, mark it as non-offloadable or employ a different scheduling policy.

4 Nanos6 runtime implementation

An OmpSs-2@Cluster application is executed in the same way as any MPI program, e.g., using `mpirun` or `mpiexec`. All processes contain a Nanos6 runtime instance, as shown in Figure 2. The nodes are peers, the only distinction among them being that node 0 executes `main` and it performs runtime operations that require internal collective synchronizations like `nanos6_dmalloc` (which may be called on any node). Each node (including node 0), creates a single "namespace" task, which is the implied common parent of all tasks offloaded to that node. The processes communicate via point-to-point MPI, with a dedicated thread on each node to handle MPI control messages.

During runtime initialization, all processes coordinate to map a common virtual memory region into their virtual address space, organized as in Figure 3. Each node owns a portion of the local memory region, so that it can allocate stacks and user data without coordinating with other nodes. Similarly, the distributed memory region is available for allocations using `nanos6_dmalloc`. Any data residing in any of these regions can be used by a task on any node. At initialization time only the virtual memory is mapped, physical memory will be allocated on demand, when accessed by a task that executes on the node. Since all regions are pre-allocated on all nodes, there is no need for temporary memory allocation, address translation or user intervention.

[1] Currently `nowait` is available through a Nanos6 API call.

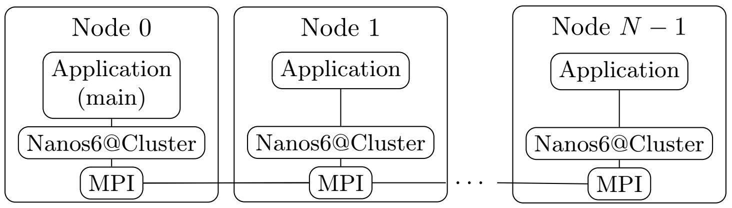


Fig. 2: OmpSs-2@Cluster architecture. Each node is a peer, except that Node 0 runs the `main` task and performs distributed memory allocations.

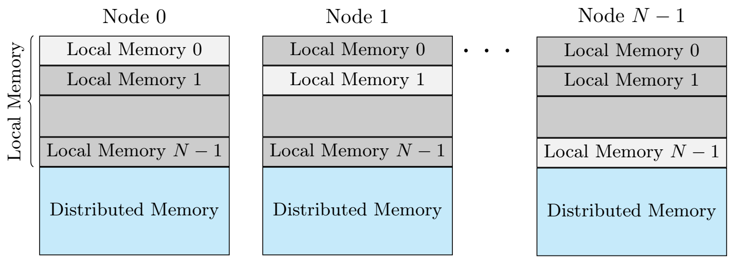


Fig. 3: Runtime memory map, which is common to all processes on all nodes.

The problem of running the whole program, which is a hierarchy of tasks with dependencies, is conceptually separated into building the distributed dependency graph (Section 4.1), tracking dependencies among tasks (4.2), scheduling ready tasks for execution (4.3), and performing data transfers before executing tasks (4.4). All, of course, happen for multiple tasks concurrently.

4.1 Building the distributed dependency graph

Tasks are created by their parent task (which may be `main`). Once tasks become ready, they are allocated to a cluster node. Many offloadable tasks have only weak dependencies, so they are ready immediately and are offloaded in advance (e.g. "weakmatvec" in Figure 1). This allows concurrent subtask creation on all nodes, overlapped with execution and optimizations to reduce the number of control messages on the critical path. If the task is offloaded, a Task New message is sent from the *creation node*, where the task was first created, to the *execution* node, where it will be executed. This is shown in steps ① and ② in Figure 4, which illustrates the execution of two offloaded tasks on different nodes sharing an `inout` dependency. The execution node uses the taskInfo information embedded in the message to create and submit the *proxy task* that will execute the task body.

A key design choice of OmpSs-2@Cluster is that no cluster node builds the whole computation graph of the application. Distributing the computation graph across cluster nodes minimizes coordination, thus allowing more potential for scalability. Nodes independently choose whether, and to which node, to offload any task created on that node. When a task is offloaded, the predecessor task for each access is known, and in many cases, its execution node is also known.

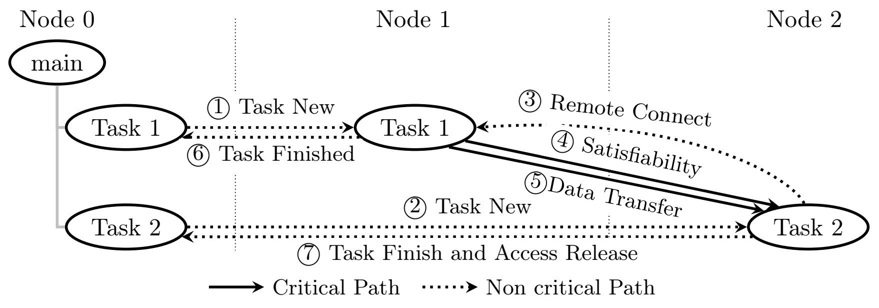


Fig. 4: Direct connection between Task 1 on Node 1 and Task 2 on Node 2. All messages are point to point.

The identity of the predecessor's execution node is passed through the dependency system as tasks are offloaded. When the task's execution node creates the proxy task, dependencies from a predecessor on the same node are connected inside the namespace task, in the same way as any other sibling tasks. If the predecessor's execution node is different, then a Remote Connect message is sent to the previous node (step ③). Messages to connect the graph are off the critical path and are distributed among the implied nodes with non-blocking point-to-point communications.

4.2 Tracking dependencies among tasks

When tasks complete, dependencies are released by sending point-to-point Satisfiability messages from the predecessor to successor node (step ④). These messages indicate which dependency regions are satisfied and their locations, and the MPI tags for associated eager data sends (see Section 4.4). If the access of an offloaded task has no successor on another node, the access is instead released back to the offloader (step ⑦). Write-after-read accesses, i.e. `inout` following multiple `in` accesses, are synchronized at the *creation node*, with satisfiability passed to the `inout` access once all the `in` accesses have released the access back.

4.3 Scheduling ready tasks for execution

When a task becomes ready, the node that created the task decides whether to offload it, and, if so, it decides which node it should be offloaded to. Currently the majority of programs have two levels of nesting: one level across nodes and one level across the cores on the node. These programs have a single level of offloadable tasks, so all offloaded tasks are created on the same node, so no coordination among nodes is required for load balancing. Future work will improve the scheduler to support distributed load balancing among nodes. The scheduling of ready tasks to be executed on a node, whether offloaded or not, is done using the normal Nanos6 scheduler. The scheduler exploits all the cores on the node, and it allows variants of tasks and/or subtasks to execute on accelerators.

4.4 Data transfers

Data transfers may be configured to be either lazy or eager. Lazy data transfers are the safest option, but delaying the initiation of the data transfer until the data is required by a strong access may cause latency on starting tasks that require data from another node. This latency is somewhat mitigated when there are sufficient tasks to keep the cores busy. Eager data transfers initiate the data transfers at the earliest moment, even for weak accesses, which is when Satisfiability is sent. The Satisfiability message (step ④) is immediately accompanied with a data transfer (step ⑤). This is generally a good optimization, but it may happen that data contained in a weak access is not accessed by any subtask with a strong access, in which case the data transfer would be unnecessary. This situation happens in Cholesky factorization, due to the pattern of the data accesses of the `dgemm` tasks. Since all subtasks are assumed to be created some time in advance, this situation can be detected when a task with weak accesses completes, at which point all subtasks have been created, but no subtask accesses all or part of the weak access. In this case a No Eager Send message is sent to the predecessor. In all cases, when a task completes, the data is not copied back to the parent (write-back) unless needed by a successor task or taskwait. The latest version of the data remains at the execution node until needed.

5 Evaluation methodology

5.1 Hardware and software platform

We evaluate OmpSs-2@Cluster on MareNostrum 4 [4]. Each node has two 24-core Intel Xeon Platinum 8160 CPUs at 2.10 GHz, for a total of 48 cores per node. Each socket has a shared 32 MB L3 cache. The HPL Rmax performance equates to 1.01 TF per socket. Communication uses Intel MPI 2018.4 over 100 Gb/s Intel OmniPath, with an HFI Silicon 100 series PCIe adaptor. The runtime and all the benchmarks were compiled with Intel Compiler 18.0.1, and all the kernels use the same code and same standard BLAS functions from Intel MKL 2018.4.

5.2 Benchmarks

We use simple and optimized variants of four benchmarks all executed in configurations of 2 processes per node (one per NUMA node) from 1 to 16 nodes for a total of 32 MPI processes:

matvec is a sequence of row cyclic matrix–vector multiplications without dependencies between iterations. This benchmark has fine-grained tasks with complexity $O(N^2)$ and no data transfers. It exposes the need to implement and improve the namespace and direct propagation approaches; as well as exhibiting patterns that require reduction and grouping of control messages. The results in Section 6 show how this benchmark performs with a simple implementation using a single level of strong tasks vs. an optimized implementation with nested weak and strong tasks.

Benchmark	MPI+ OpenMP		OmpSs-2@Cluster Simple		OmpSs-2@Cluster Optimized	
matvec	MPI point-to-point, OMP parallel for	10	One level strong tasks	11	Nested weak and strong tasks	18
matmul	MPI point-to-point, OMP parallel for	10	Nested only strong tasks	18	Nested weak and strong tasks	18
jacobi	MPI collective gather, OMP parallel for	20	Task for	25	Nested weak and strong tasks with wait clause	30
cholesky	MPI point-to-point OMP Tasks	135	strong tasks	30	Task for, memory reordering and priority	134

Table 2: Benchmark characteristics and number of lines of code of kernels.

matmul is a matrix–matrix multiplication performed to study the behavior with bigger tasks of $O(N^3)$ with a similar access pattern. This benchmark was useful to detect redundant unneeded data transfers negligible with matvec. In this case we compare a simple version with nested strong tasks vs. nested weak and strong tasks to compare the impact of early offloading vs access fragmentation consequence of early release without data transfers or wait clause.

jacobi is an iterative Jacobi solver for strictly diagonally dominant systems. It has the same $O(N^2)$ complexity as matvec, but it has $(N-1)^2$ data transfers between iterations. The objective was to measure the impact of data transfers and control messages to detect optimization opportunities. With this benchmark we detected fragmentation associated with early release and therefore implemented the autowait feature. The simple version uses `task for` (simpler) and the optimized version adds a helper task to reduce control messages and fragmentation.

cholesky is a Cholesky factorization with a complex execution and dependencies pattern. This benchmark performs a higher number of smaller tasks, compared with matmul, and it introduces load imbalance and irregular patterns. The simple version code uses strong tasks and only needs few lines, while the optimized version uses `task for` and memory reordering optimizations to reduce fragmentation and data transfers.

All MPI+OpenMP versions were implemented in two variants, using `parallel for` and OpenMP tasks. The best version was selected and reported in Section 6.

Table 2 shows the key benchmark characteristics and the number of lines of code for all implementations. The values consider only the computational parts, ignoring initialization, range specific code and conditions needed in MPI and not required in OmpSs, they also exclude timing, prints and comments. We see that matmul, matvec and jacobi are all small kernels, with no major differences in size. The OmpSs-2@Cluster versions of some of these small benchmarks are larger, due to the enclosing weak task (see Figure 1), at little increase in complexity.

6 Results

Figure 5 shows the strong scaling results for all the benchmarks. Every experiment was executed 10 times; `jacobi` and `matvec` with 400 iterations each. All points in the graphs include error bars, but in most cases they are hard to see as the standard deviation is usually insignificant.

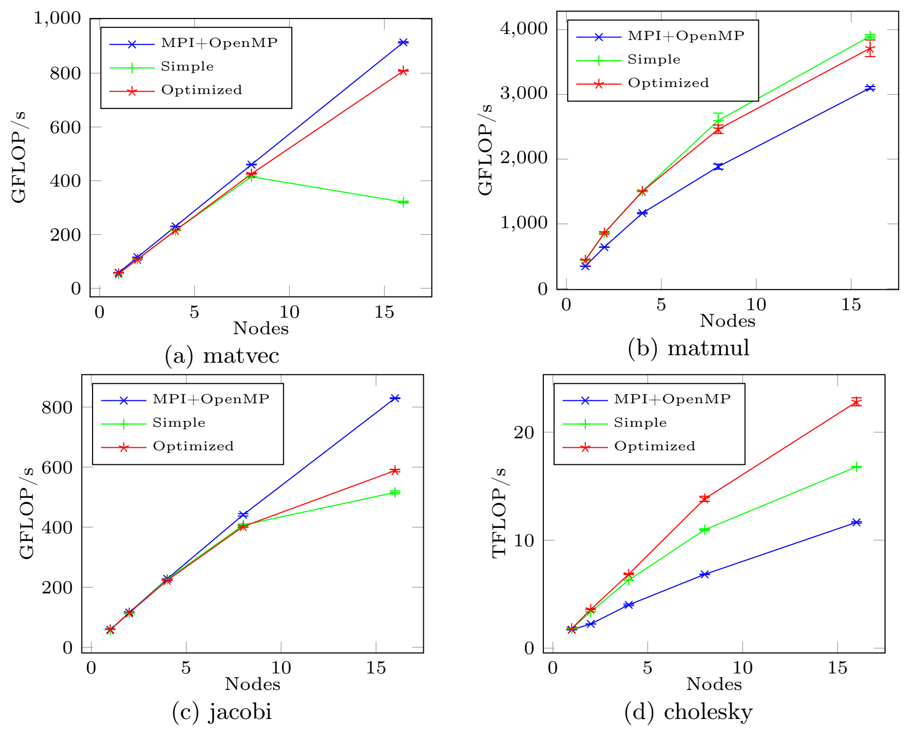
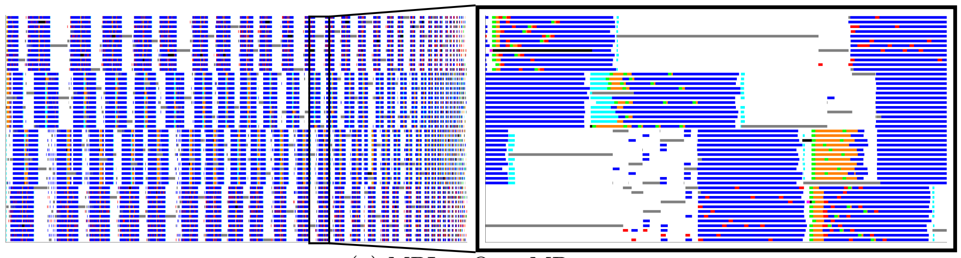


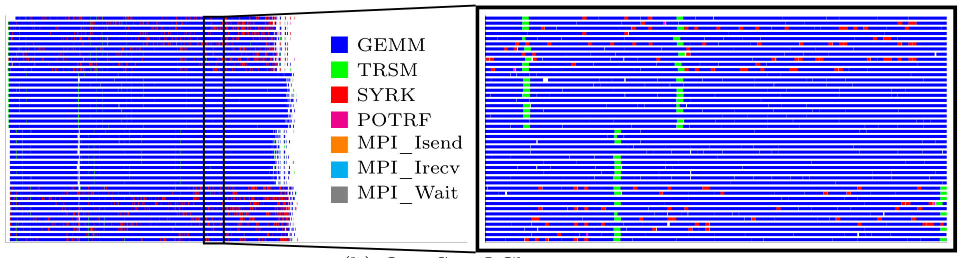
Fig. 5: Strong scaling for OmpSs-2@Cluster and MPI+OpenMP. The x-axis is the number of nodes and the y-axis is the compute throughput in GFLOP/sec.

In all subplots, the x-axis is the number of nodes, from 1 to 16 (see Section 5.2), and the y-axis is the performance. We see that the simple (unoptimized) OmpSs-2@Cluster matvec, matmul and jacobi have similar performance to MPI on up to 8 nodes. The optimized code for matvec and matmul has somewhat better performance and is similar to MPI up to 16 nodes. These two benchmarks perform multiple iterations, and do not require data transfers between iterations, so they evaluate the impact of task offload and enforcing of dependencies. On the other hand, jacobi has all-to-all communication, which is optimized using a collective in the MPI implementation but is done with point-to-point transfers by Nanos6. This limits the scalability and is an avenue for future research.

Finally, cholesky has a more complex communication and dependency pattern. Due to asynchronous tasking and early release of dependencies, the OmpSs-2@Cluster implementation achieves better performance than MPI+OpenMP tasks, and it has twice its performance on 16 nodes. The simple OmpSs-2@Cluster implementation of cholesky achieves better performance than MPI + OpenMP with a 4.5× reduction in code size. The optimized OmpSs-2@Cluster code for cholesky has similar length to the MPI+OpenMP version as shown in Table.2.



(a) MPI + OpenMP



(b) OmpSs-2@Cluster

Fig. 6: Paraver/Extrae traces showing synchronization for cholesky: 16384×16384 matrix on four nodes, 12 cores per node

To understand this in more detail, Figure 6 compares Paraver/Extrae traces for cholesky with MPI + OpenMP vs. optimized OmpSs-2@Cluster. To show an intelligible trace, it is a small example of a 16384×16384 matrix on four nodes, with 12 cores per node. Both traces show a time lapse of 1160 ms since the algorithm start; the zoomed regions are 50 ms. The traces show the BLAS kernels and MPI communications (only non-negligible in the MPI version).

We see that the OmpSs-2@Cluster version has almost 100% utilization, but synchronization among tasks and MPI communication causes the MPI+OpenMP version to have a utilization of only about 50%. As MPI is not task-aware, there are limitations on how MPI calls may be used with OpenMP Tasks to avoid deadlock. Otherwise, tasks in all threads may try to execute blocking MPI calls, occupying all threads even though other tasks may be ready, leading to a deadlock. Resolving this needs synchronization, such as serializing waits or limiting the number of send or receive tasks in every step with artificial dependencies. Figure 6a shows how the waits (gray) stop parallelism between iterations.

On the other hand, Figure 6b does not show any waits because all communication in OmpSs-2@Cluster is non-blocking. Tasks not satisfied are not ready, so they remain in the dependency system; while ready tasks with pending data transfers, are re-scheduled when they can execute. This approach allows the runtime to concurrently execute tasks from multiple iterations and keep the workers busy while transfers occur. The priority clause is advantageous to prioritize the scheduling of critical-path tasks, in a similar way to OmpSs-2 on SMP, but it is more important for OmpSs-2@Cluster due to the network latency.

Programming model	*Languages*			*Task definition*		*Task discovery*		*Nested tasks*		*Address space*		*Dependencies*		*Depend. graph*			*Location tracking*		
	C	C++	Fortran	From sequential	Explicit	Static	Dynamic	Supported	Offloadable	Common	Different	Flexible	Early release	Distributed	Centralized	Duplicated	Via dependencies	Directory-based	Owner computes
OmpSs-2@Cluster	✓	✓	✓	✓	✓		✓	✓	✓	✓		✓	✓	✓			✓		
OmpSs-1@Cluster	✓	✓	✓	✓	✓		✓	✓	?		✓				✓			✓	
StarPU-MPI	✓	✓			✓		✓	✓			✓	?				✓			✓
DuctTeip		✓			✓		✓	✓			✓					✓		✓	
DASH		✓			✓		✓	✓		✓						✓			✓
PaRSEC	✓	?	✓		✓	✓	✓	?	?		✓					✓			✓
Charm++	✓	✓	✓		✓		✓		✓		✓								

Table 3: Comparison of distributed tasking models

7 Related work

Shared memory: Numerous frameworks support shared memory task parallelism with dependencies, with the *de facto* standard being OpenMP version 4.0 or above [23]. **Cilk** [12] is perhaps the first well known task-based programming model, which identifies tasks with the `spawn` keyword and supports synchronization using the `sync` statement. **Cilk++** [21] adds support for parallel loops. **XKappi** [34] also has a directed acyclic graph of tasks like OpenMP. **Wool** [16] is a low overhead library for nested tasks.

Partee [24] is an OmpSs programming model alternative implementation using **BDDT** [33] with strong effort to better handle more fine-grained tasks and irregular dependencies. **Raja** [11] and its associated libraries provide a similar C++11 approach to performance portable programming.

Distributed memory: Table 3 summarizes the main frameworks for fine-grained distributed memory task parallelism. Most of them wrap or extend existing frameworks for shared memory tasking.

OmpSs-1@Cluster is a variant of OmpSs-1 for clusters of GPUs [13]. It has a similar approach to OmpSs-2@Cluster, but task creation and submission is centralized, dependencies are only among sibling tasks, and address translation is needed for all task accesses. It uses a directory on one node to track all data location, rather than passing the location through the edges of a distributed dependency graph. It has only strong tasks, so it has limited ability to overlap execution with task creation overhead.

StarPU-MPI [3] is the multi-node extension of StarPU. In this model all processes create the same graph of top-level tasks and it uses an owner-computes model as shown in Table 3. Task allocation to nodes and communications for data transfers are at task creation. Posting the receives in advance removes the need of Satisfiability messages, but it implies high memory consumption and some throttling mechanism which limits parallelism discovery [31].

DuctTeip [35] is a distributed task-parallel framework implemented on top of SuperGlue [32]. This approach based on data versioning supports general task graphs to implement common application structures. It divides the computation into levels to build the task graph in parallel. Child tasks are created like the strong tasks of OmpSs-2, so task creation could be on the critical path.

CHAMELEON [20] is a library for fine-grained load-balancing in task-parallel MPI+X applications. The implementation is optimized for responsiveness to changing execution conditions. They do not optimize for strong scaling of task offloading. Data is always copied back to the parent after the task, and it uses a collective distributed taskwait and the OpenMP target construct. **PaRSEC** [18] is a platform for distributed task execution. A front-end compiler generates a parameterized Directed Acyclic Graph (DAG). Tasks can be generated dynamically and could be submitted by other tasks. Dependencies are not just of the DAG type, but they support nesting, concurrent and commutative dependencies, and `weak` and `strong` dependencies. The tasks' data can be described by region dependencies with fragmentation. **DASH** [17] is a C++ template library that extends C++ STL concepts to distributed memory. It is based on a PGAS-based distributed task programming approach. Every process creates a local dependency graph in parallel. The dependencies on non-local memory are automatically resolved by the runtime system. The execution is divided into phases because there is no total ordering on the dependencies among nodes. **Charm++** [25] is a C++-based object oriented programming model for running migratable objects known as "chares". It uses a message-driven runtime model in which methods on chares result in sending a message to the chare, resulting in asynchronous function execution with similarities to task execution.

Legion [10] is a parallel programming system with an OOP syntax similar to C++ based on logical regions to describe the organization of data, with an emphasis on locality and task nesting via an object-oriented syntax. Part of Legion's low-level runtime system uses UPC's GASNet. Other approaches include **COMPSs** [22], which is a Java, C/C++ and Python framework to run parallel applications on clusters, clouds and containerized platforms. The execution granularity is much coarser with data transfer via files. **Pegasus** [15] is another workflow management system that uses a DAG of tasks and dependencies. **GPI-Space** [28] 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. **HPX** [19] is an implementation of the ParalleX programming paradigm, with an Active Global Address Space (AGAS) to manage the locality of global objects. **X10** [14] is an object-oriented programming language for high-productivity programming that spawns asynchronous computations, with the programmer responsible for PGAS data distribution.

8 Acknowledgements and data availability

The datasets and code generated during and/or analyzed during the current study are available in the Figshare repository: [1]. 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 754337 (EuroEXA). It is supported by the Spanish State Research Agency - Ministry of Science and Innovation (contract PID2019-107255GB and Ramon y Cajal fellowship RYC2018-025628-I) and by the Generalitat de Catalunya (2017-SGR-1414).

9 Conclusions

This paper presented OmpSs-2@Cluster, a programming model and runtime system, which provides efficient support for hierarchical tasking from distributed memory to threads. We describe the programming model and runtime optimizations to build the distributed dependency graph, enforce dependencies, perform eager data fetches and execute tasks.

The results show that performance of OmpSs-2@Cluster is competitive with MPI+OpenMP for regular and well-balanced applications. For irregular or unbalanced applications it may be significantly better without increasing the code complexity or sacrificing the programmer productivity. This work opens future work to leverage this model for (a) resiliency, (b) scalability, (c) intelligent multi-node load balancing, and (d) malleability. With this aim, all source code and examples are available open source [9].

References

1. Aguilar Mena, J., Shaaban, O., Beltran, V., Carpenter, P., Ayguade, E., Labarta Mancho, J.: Artifact and instructions to generate experimental results for the Euro-Par 2022 paper: "OmpSs-2@Cluster: Distributed memory execution of nested OpenMP-style tasks" (2022). https://doi.org/10.6084/m9.figshare.19960721
2. Álvarez, D., Sala, K., Maroñas, M., Roca, A., Beltran, V.: Advanced Synchronization Techniques for Task-Based Runtime Systems, p. 334–347. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3437801.3441601
3. Augonnet, C., Aumage, O., Furmento, N., Namyst, R., Thibault, S.: StarPU-MPI: Task programming over clusters of machines enhanced with accelerators. In: European MPI Users' Group Meeting. pp. 298–299. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-33518-1_40
4. Barcelona Supercomputing Center: MareNostrum 4 (2017) System Architecture (2017), `https://www.bsc.es/marenostrum/marenostrum/technical-information`
5. Barcelona Supercomputing Center: Mercurium (2021), `https://pm.bsc.es/mcxx`
6. Barcelona Supercomputing Center: Nanos6 (2021), `https://github.com/bsc-pm/nanos6`
7. Barcelona Supercomputing Center: OmpSs-2 releases (2021), `https://github.com/bsc-pm/ompss-releases`
8. Barcelona Supercomputing Center: OmpSs-2 specification (2021), `https://pm.bsc.es/ftp/ompss-2/doc/spec/`

9. Barcelona Supercomputing Center: OmpSs-2@Cluster releases (2022), `https://github.com/bsc-pm/ompss-2-cluster-releases`
10. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 1–11 (2012). https://doi.org/10.1109/SC.2012.71
11. Beckingsale, D.A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A.J., Pearce, O., Robinson, P., Ryujin, B.S., Scogland, T.R.: Raja: Portable performance for large-scale scientific applications. In: IEEE/ACM international workshop on performance, portability and productivity in HPC (P3HPC) (2019). https://doi.org/10.1109/P3HPC49587.2019.00012
12. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Journal of Parallel and Distributed Computing **37** (02 1999). https://doi.org/10.1006/jpdc.1996.0107
13. Bueno, J., Planas, J., Duran, A., Badia, R.M., Martorell, X., Ayguade, E., Labarta, J.: Productive programming of GPU clusters with OmpSs. In: IEEE 26th International Parallel and Distributed Processing Symposium (5 2012). https://doi.org/10.1109/IPDPS.2012.58
14. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: 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. p. 519–538. OOPSLA '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1094811.1094852
15. Deelman, E., Singh, G., Su, M.H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G., Good, J., Laity, A., Katz, D.S.: Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Scientific Programming **13**(3), 219–237 (01 2005). https://doi.org/10.1155/2005/128026
16. Faxén, K.F.: Wool user's guide. Tech. rep., Technical report, Swedish Institute of Computer Science (2009)
17. Fürlinger, K., Gracia, J., Knüpfer, A., Fuchs, T., Hünich, D., Jungblut, P., Kowalewski, R., Schuchart, J.: DASH: Distributed data structures and parallel algorithms in a global address space. In: Software for Exascale Computing-SPPEXA 2016-2019. pp. 103–142. Springer International Publishing (07 2020). https://doi.org/10.1007/978-3-030-47956-5_6
18. Hoque, R., Herault, T., Bosilca, G., Dongarra, J.: 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. pp. 1–8 (11 2017). https://doi.org/10.1145/3148226.3148233
19. Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D.: HPX: A task based programming model in a global address space. In: 8th International Conference on Partitioned Global Address Space Programming Models (2014). https://doi.org/10.13140/2.1.2635.5204
20. Klinkenberg, J., Samfass, P., Bader, M., Terboven, C., Müller, M.: CHAMELEON: Reactive load balancing for hybrid MPI+OpenMP task-parallel applications. Journal of Parallel and Distributed Computing **138** (12 2019). https://doi.org/10.1016/j.jpdc.2019.12.005
21. Leiserson, C.E.: The Cilk++ concurrency platform. The Journal of Supercomputing **51**(3), 244–257 (2010). https://doi.org/10.1007/s11227-010-0405-3

22. Lordan, F., Tejedor, E., Ejarque, J., Rafanell, R., Alvarez, J., Marozzo, F., Lezzi, D., Sirvent, R., Talia, D., Badia, R.M.: ServiceSs: An interoperable programming framework for the cloud. Journal of grid computing **12**(1) (2014). https://doi.org/10.1007/s10723-013-9272-5
23. OpenMP Architecture Review Board: OpenMP 4.0 complete specifications (July 2013)
24. Papakonstantinou, N., Zakkak, F.S., Pratikakis, P.: Hierarchical parallel dynamic dependence analysis for recursively task-parallel programs. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 933–942 (2016). https://doi.org/10.1109/IPDPS.2016.53
25. Parallel Programming Lab, Dept of Computer Science, U.o.I.: Charm++ documentation, `https://charm.readthedocs.io/en/latest/index.html`
26. Perez, J.M., Beltran, V., Labarta, J., Ayguadé, E.: Improving the integration of task nesting and dependencies in OpenMP. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 809–818 (5 2017). https://doi.org/10.1109/IPDPS.2017.69
27. Pérez, J., Badia, R.M., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In: Proceedings - IEEE International Conference on Cluster Computing, ICCC. pp. 142–151 (09 2008). https://doi.org/10.1109/CLUSTR.2008.4663765
28. Rotaru, T., Rahn, M., Pfreundt, F.J.: MapReduce in GPI-Space. In: Euro-Par 2013: Parallel Processing Workshops. pp. 43–52. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54420-0_5
29. Sala, K., Macià, S., Beltran, V.: Combining one-sided communications with task-based programming models. In: 2021 IEEE International Conference on Cluster Computing (CLUSTER). pp. 528–541 (2021). https://doi.org/10.1109/Cluster48925.2021.00024
30. Sala, K., Teruel, X., Perez, J.M., Peña, A.J., Beltran, V., Labarta, J.: Integrating blocking and non-blocking MPI primitives with task-based programming models. Parallel Computing **85**, 153–166 (2019). https://doi.org/10.1016/j.parco.2018.12.008
31. Sergent, M., Goudin, D., Thibault, S., Aumage, O.: Controlling the memory subscription of distributed applications with a task-based runtime system. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 318–327 (2016). https://doi.org/10.1109/IPDPSW.2016.105
32. Tillenius, M.: SuperGlue: A shared memory framework using data versioning for dependency-aware task-based parallelization. SIAM Journal on Scientific Computing **37**(6) (2015). https://doi.org/10.1137/140989716
33. Tzenakis, G., Papatriantafyllou, A., Kesapides, J., Pratikakis, P., Vandierendonck, H., Nikolopoulos, D.S.: BDDT: Block-level dynamic dependence analysisfor deterministic task-based parallelism. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '12, vol. 47, pp. 301–302. Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2145816.2145864
34. Virouleau, P., Broquedis, F., Gautier, T., Rastello, F.: Using data dependencies to improve task-based scheduling strategies on NUMA architectures. In: European Conference on Parallel Processing. pp. 531–544. Springer (2016). https://doi.org/10.1007/978-3-319-43659-3_39
35. Zafari, A., Larsson, E., Tillenius, M.: DuctTeip: An efficient programming model for distributed task-based parallel computing. Parallel Computing (2019). https://doi.org/10.1016/j.parco.2019.102582