Automatic aggregation of subtask accesses for nested OpenMP-style tasks

1st Omar Shaaban
Barcelona Supercomputing Center
Barcelona, Spain
omar.ibrahim@bsc.es

2nd Jimmy Aguilar
Barcelona Supercomputing Center
Barcelona, Spain
jaguilar@bsc.es

3rd Vicenç Beltran
Barcelona Supercomputing Center
Barcelona, Spain
vicenc.beltran@bsc.es

4th Paul Carpenter
Barcelona Supercomputing Center
Barcelona, Spain
paul.carpenter@bsc.es

5th Eduard Ayguadé
Barcelona Supercomputing Center
Barcelona, Spain
eduard.ayguade@bsc.es

6th Jesus Labarta Mancho
Barcelona Supercomputing Center
Barcelona, Spain
jesus.labarta@bsc.es

Abstract*—Task-based programming is a high performance and productive model to express parallelism. Tasks encapsulate work to be executed across multiple cores or offloaded to GPUs, FPGAs, other accelerators or other nodes. In order to maintain parallelism and afford maximum freedom to the scheduler, the task dependency graph should be created in parallel and well in advance of task execution. A key limitation with OpenMP and OmpSs-2 tasking is that a task cannot be created until all its accesses and its descendents' accesses are known. Current approaches to work around this limitation either stop task creation and execution using a taskwait or they substitute "fake" accesses known as sentinels. This paper proposes the `auto` clause, which indicates that the task may create subtasks that access unspecified memory regions or it may allocate and return memory at addresses that are of course not yet known. Unlike approaches using taskwaits, there is no interruption to the concurrent creation and execution of tasks, maintaining parallelism and the scheduler's ability to optimize load balance and data locality. Unlike existing approaches using sentinels, all tasks can be given a precise specification of their own data accesses, so that a single mechanism is used to control task ordering, program data transfers on distributed memory and optimize data locality, e.g. on NUMA systems. The `auto` clause also provides an incremental path to develop programs with nested tasks, by removing the need for every parent task to have a complete specification of the accesses of its descendent tasks. This is redundant information that can be time consuming and error-prone to describe. We present a straightforward runtime implementation that achieves a 1.4 times speedup for n-body with OmpSs-2@Cluster task offloading to 32 nodes and $<4\%$ slowdown for three benchmarks with task offloading to 8 nodes. All code is open source.

***Index Terms*—HPC, Runtime Systems, OpenMP, OmpSs-2, Task Parallelism, Distributed Computing**

I. Introduction

Task-based programming is a high performance and productive model for regular and irregular parallel execution [1]. Tasks encapsulate work to be executed concurrently across the CPU cores, passed to accelerators such as GPUs [2] or FPGAs [3], or even offloaded to other nodes [4]–[8]. In OpenMP and OmpSs-2, tasks are defined by an annotated sequential program. The sequential semantics gives a clear and familiar meaning to the program and it simplifies the porting of existing codes. Each task is identified using a pragma annotation, which indicates the regions of memory accessed by the task and whether each access is read-only, write-only or read–write. These data accesses are used to: (1) compute dependencies to control ordering of tasks, (2) program data transfers if necessary, and (3) schedule tasks close to their data, e.g. on different NUMA domains.

As systems scale to larger numbers of cores and/or accelerators, it becomes infeasible for one sequential thread to create enough tasks to keep all the cores busy. The natural solution is to build the full dependency graph in parallel, by multiple concurrently-executing parent tasks. This approach is known as task nesting, and it is supported by OpenMP [9], [10]. In OpenMP, the dependency graphs of different tasks are isolated from each other. OmpSs-2 improves the situation through its support for fine-grained dependencies among nesting levels [11]: computation of the dependencies from the data accesses is done locally inside a task, but all fine-grained tasks in the program become part of a single hierarchical dataflow dependency graph. This allows dependencies to be discovered among subtasks of different parents.

A key limitation of current OpenMP and OmpSs-2 tasking models is that a task cannot be created until the addresses and sizes of all its accesses are known. Since the task's accesses must cover all its descendents' accesses, all the accesses of its subtasks, their subtasks, and so on, also need to be inside specified regions. We illustrate the problem with a hypermatrix multiplication followed by a Cholesky decomposition. This example is shown in Figure 1, which is described in detail in Section III-A. In brief, the blocks of the hypermatrix are allocated in parallel by multiple parent tasks (`row`), so there is no good way to specify fine-grained dependencies between the matrix multiplication (`matmul`) and the Cholesky decomposition (`potrf` and others).

Existing approaches work around this problem. One way is to add synchronization to wait for the earlier tasks to complete, i.e., using a taskwait (Figure 1a). Another way is to substitute the true data access with a "fake" one that happens to imply the correct task ordering but does not provide correct information for data transfers or data locality, i.e., using a sentinel such as the pointer to the block rather than the block itself (Figure 1b).

This paper introduces the `auto` data access clause, which indicates that a task may have additional data accesses beyond those listed explicitly in the pragma annotation. These accesses may be unknowable when the task is created, either because the task allocates and returns new memory regions (e.g. `row` in Figure 1c) or because it creates subtasks that access regions defined by previous tasks (e.g. `weakpotrf` in Figure 1c). In addition, the programmer may decide not to explicitly specify all the accesses on behalf of the descendents because they are complicated to determine or express (e.g. `parent` in Figure 2b). In all these cases, the task is created and inserted into the dependency system, and it is not blocked from execution. The missing data accesses will be inferred by the runtime system and will become ordinary fine-grained accesses of the descendent tasks.

Unlike approaches using a taskwait, there is no interruption to the concurrent creation and execution of tasks, maintaining parallelism and affording maximum freedom to the scheduler to optimize load balance and data locality. Unlike approaches using sentinels, all tasks can be given a precise specification of their strong data accesses. The full specification of task data accesses means that a single mechanism is used to compute the dependencies that enforce ordering among tasks, to program data transfers on distributed memory, and to optimize data locality on NUMA and distributed memory systems. Compared with a version using sentinels, program clarity is improved, since the pragma annotations match the program's actual data accesses. It avoids the fragility of sentinels, because part of the program can be modified, e.g., the `matmul` and/or `potrf` tasks can be independently subdivided into finer-grained tasks, without having to redesign the use of sentinels throughout the whole program. By precisely specifying every task's true data accesses, the program is also suitable for task offloading on distributed memory systems.

The `auto` data access clause also provides an incremental path to develop programs with nested tasks. Since `auto` declares that the data access annotations may not cover all accesses of descendent tasks, it becomes only necessary to annotate the strong accesses for the task itself. For some applications, such as dense linear algebra, it may be straightforward to specify the precise weak accesses, but for applications involving graphs or trees, this task is more difficult. Moreover, these weak access annotations are redundant and error-prone.

While it may seem like the implementation would be complicated and expensive, in terms of overhead, we present a simple approach, with a few key optimizations, that allows efficient task execution, even in the context of task offloading to other nodes. Since the `auto` pragma aggregates information that is already given to the runtime system, there is no need for sophisticated compiler or instruction-level analysis. In many cases the overhead is negligible, though, of course, when `auto` is used indiscrimately, performance analysis may show overhead or serialization that can be avoided by specifying the data accesses of just a subset of the parent tasks.

We evaluate the performance improvement using the hypermatrix example on SMP and an n-body application with task offloading via OmpSs-2@Cluster across up to 32 nodes. By eliminating the taskwait and enabling greater overlap of the construction of the dependency graph with task execution, we get a 1.4 times higher throughput, compared with a version using taskwait. We also demonstrate how the programmer can obtain an initial functional implementation using nested tasks with the `auto` keyword. The `auto` clause gives initial versions of the `matmul-smp` and n`-body-smp` benchmarks [12] that use weak accesses, which are within 19.6% and 10% respectively of the standard version of the benchmark. Since this overhead is low, we include more challenging benchmarking using OmpSs-2@Cluster. We obtain initial versions of the OmpSs-2@Cluster `matvec`, `matmul` and `jacobi` benchmarks [7] that scale up to 8 nodes with less than 3%, 7%, and 9% degradation respectively compared with manually-specified data accesses. The `cholesky` benchmark scales to just 2 nodes, with 4.5% degradation, but Extrae/Paraver performance analysis shows a bottleneck on a single parent task. Manually specifying the data accesses on that task, and leaving `auto` accesses on the other two tasks, allows scaling up to 8 nodes.

The main contributions of this paper are:

- We propose the `auto` clause for tasks. The `auto` clause automatically aggregates all descendent accesses and memory allocations into weak accesses. We also propose the `none` clause, which excludes a region of memory from analysis and may sometimes avoid serialization.
- We describe two use cases for the `auto` clause: (1) memory allocation inside tasks and (2) programmer productivity through automatic aggregation of subtask accesses.
- We motivate and evaluate the approach using the hypermatrix example and an n-body application with a tree of size unknown at task creation time, and with two SMP benchmarks and four distributed memory benchmarks for programmer productivity. We show the benefits of the approach even in the context of task offloading to other nodes.

We release all code open source [13], in the hope of fostering uptake in applications. A strong advantage of this proposal is that there is a simple baseline implementation. Nevertheless, future work may succeed in further reducing overhead, while presenting the same straightforward programmer's model.

II. Background

A. *OmpSs-2, Nanos6 and Mercurium*

OmpSs-2 [14] is the second generation of the OmpSs parallel programming model. It is developed as a research platform to explore and demonstrate features for potential future standardization in OpenMP [15]. OmpSs-2 is a dataflow centric model similar to OpenMP, with tasking, dependencies, and heterogeneity, integrated with compiler directives for C++ and FORTRAN. Unlike OpenMP, OmpSs-2 uses a thread-pool execution model, it targets heterogeneous architectures via native kernels, and it enables concurrency through asynchronous parallelism. In OmpSs-2, task data accesses are used as a single mechanism to compute dependencies that enforce task ordering, to determine data locality and, when necessary, to program data copies.

OmpSs-2 extends OmpSs and OpenMP to improve task nesting and fine-grained dependencies across multiple nesting levels [11], [16]. The `depend` clause is extended to support the `weakin`, `weakinout` and `weakout` access types, which indicate that the task itself does not access the data, but its nested subtasks may do so. Any subtask that directly accesses data needs to specify a dependency with a strong (non-weak) access type. Any task that delegates the access to a subtask must include the data region in at least the weak variant. Weak accesses provide a linking point between the dependency domains at different nesting levels, but they do not delay parent task execution or require data transfers on distributed memory. The addition of weak dependencies exposes more parallelism, allows better scheduling decisions and it enables parallel instantiation of tasks with dependencies among them.

The annotated program is translated by the source-to-source Mercurium [17] compiler into one that calls the Nanos6 [18] runtime API. The runtime is responsible for computing task dependencies from the accesses, and it schedules and executes tasks, respecting the implied task dependency constraints and performing data transfers and synchronizations.

B. OmpSs-2@Cluster

OmpSs-2@Cluster [7], [13] is the task offloading extension of OmpSs-2. OmpSs-2@Cluster can be used as an alternative to MPI for small-scale parallelism or as a way to mitigate load imbalance in MPI+OmpSs-2 programs, by automatically offloading tasks from heavily-loaded nodes to balance the work [19]. Underlying communication for control messages and data transfers is done using MPI.

Any program with OmpSs-2 tasks and a full specification of task accesses is compatible with OmpSs-2@Cluster. It is only necessary to enable task offloading in the configuration file passed to a cluster-compatible build of Nanos6. All ranks have the same virtual address space, so data allocated on one node can be seamlessly accessed by tasks that execute on any other node [7]. This avoids address translation and allows direct use of existing data structures with pointers. Nested tasks are particularly important for OmpSs-2@Cluster, as they make it possible for the nodes to concurrently create local fine-grained tasks to occupy all the cores.

Each node runs an instance of Nanos6@Cluster, which coordinate as peers. The instances of the runtime system overlap construction of a distributed dependency graph, enforcing of dependencies, scheduling, data transfers, and task execution. Data copies are done for tasks or taskwaits as required, with no automatic write-back to the original node unless the data value is actually needed.

```
typedef double Block[TS][TS];
Block *A[NT][NT], *B[NT][NT], *C[NT][NT];
// Matrix multiplication
for(int i=0; i<NT; i++) {
  #pragma oss task depend(out: C[i][0;NT]) label("row")
  for(int j=i; j<NT; j++) {
    C[i][j] = calloc(1, sizeof(Block));
    for(int k=0; k<NT; k++) {
      if (A[i][k] && B[k][j]) {
        #pragma oss task depend(inout: *C[i][j]) label("matmul")
        dgemm(A[i][k], B[k][j], C[i][j]);
      }
    }
  }
}
#pragma oss taskwait
// Cholesky decomposition
for (int k=0; k<NT; k++) {
  if (C[k][k]) {
    #pragma oss task depend(inout: *C[k][k]) label("potrf")
    potrf(C[k][k]);
  }
  // Rest of Cholesky
}
```

(a) Approach with additional synchronization (taskwait)

```
// Matrix multiplication
for(int i=0; i<NT; i++) {
  #pragma oss task depend(out: C[i][0;NT]) label("row")
  for(int j=i; j<NT; j++) {
    C[i][j] = calloc(1, sizeof(Block));
    for(int k=0; k<NT; k++) {
      if (A[i][k] && B[k][j]) {
        #pragma oss task depend(inout: C[i][j]) label("matmul")
        dgemm(A[i][k], B[k][j], C[i][j]);
      }
    }
  }
}
// Cholesky decomposition
for (int k=0; k<NT; k++) {
  #pragma oss task depend(inout: C[k][k]) label("potrf")
  if (C[k][k]) {
    potrf(C[k][k]);
  }
  // Rest of Cholesky
}
```

(b) Approach with a "fake dependency" (sentinel)

```
// Matrix multiplication
for(int i=0; i<NT; i++) {
  #pragma oss task depend(out: C[i][0;NT]) depend(auto) depend(none:
        C[0;NT][0;NT]) label("row")
  for(int j=i; j<NT; j++) {
    C[i][j] = calloc(1, sizeof(Block));
    for(int k=0; k<NT; k++) {
      if (A[i][k] && B[k][j]) {
        #pragma oss task depend(inout: *C[i][j]) label("matmul")
        dgemm(A[i][k], B[k][j], C[i][j]);
      }
    }
  }
}
// Cholesky decomposition
for (int k=0; k<NT; k++) {
  #pragma oss task depend(in: C[k][k]) depend(auto) label("weakpotrf")
    if (C[k][k]) {
      #pragma oss task depend(inout: *C[k][k]) label("potrf"))
      potrf(C[k][k]);
    }
  }
  // Rest of Cholesky
}
```

(c) Approach with proposed `auto` data accesses

Fig. 1: Hypermatrix multiply followed by Cholesky decomposition. The conventional approaches either (a) involve extra synchronization or (b) obscure the data accesses. Approach (c) using `auto` is clearer, it provides locality information to the runtime, and is suitable for distributed memory.

III. Motivation

A. *Precise specification of data accesses without taskwaits*

Figure 1 shows three versions of an example program that performs a matrix multiplication to calculate the upper triangle of symmetric $C = AB$ followed by a Cholesky decomposition of C. The three matrices are stored as hypermatrices, i.e., element `A[i][j]`, `B[i][j]` or `C[i][j]` is either `NULL` or a pointer to the actual block. For simplicity, only the data accesses involving matrix C are shown. We identify tasks using the OmpSs-2 `label` clause, which provides a string literal that can be used by a performance or debugger tool to identify the tasks in a human-readable format. In the spirit of building the dependency graph in parallel, the tasks in each row of matrix C are created by a different parent task, labelled `row`.

Figure 1a has a precise specification of the data accesses of the `matmul` and `potrf` tasks, but it requires the taskwait on line 16. There is otherwise no way to connect the dependency on the actual blocks, written `*C[i][j]`, from `matmul` to `potrf`. The taskwait is needed to ensure that the main program creates the `potrf` tasks with the `*C[i][j]` access at the correct address allocated on line 7, and that these tasks cannot be executed until the block has been written. The accesses to `*C[i][j]` are still required: to serialize the `matmul` tasks that contribute to the same block of C and to manage the dependencies among the tasks performing the Cholesky decomposition (only `potrf` is shown).

The taskwait cannot be avoided by nesting `potrf` inside a parent task that has a strong dependency on `C[i][j]`. While this would correctly delay the creation of `potrf` and materialise its access to `*C[i][j]` at the right address, it will not enforce any dependencies between `matmul` and `potrf`. This is because these dependencies can only be linked if both their parents, `row` and the parent of `potrf`, have data accesses to `*C[i][j]` in at least a weak variant. We have simply moved the original problem, the data access to `*C[i][j]` at an unknown address, from `potrf` to its parent. Adding the `wait` clause to `row`, which disables early release would work, but it would delay the execution of `potrf` until a whole row of C has been written. Moreover, it is complex even for this simple example, the dependence is easy to miss, and the resulting code would be obscure and error prone.

Figure 1b shows a conventional solution using a sentinel. We replace each access on the block `*C[i][j]` with an access on the pointer to the block, `C[i][j]`. Tasks `matmul` and `potrf` have `inout` accesses on the pointer `C[i][j]`, not because they modify the pointer but because they modify the block that it points to. Although sentinels are commonly used in this way, there are three problems with this approach. First, it breaks the idea that data accesses are a unified method to specify ordering constraints, data affinity and data transfers. Since the data accesses are "fake", only for correctly enforcing ordering constraints, the runtime cannot optimize data locality properly. In particular, the runtime cannot take account of NUMA affinity on the block, `*C[i][j]` when scheduling `matmul` and `potrf`. The program also becomes unsuitable for task offloading via OmpSs-2@Cluster. Secondly, the direct connection between the pragma annotations and the behaviour of the task is broken, reducing clarity. Thirdly, if the `matmul` and/or `potrf` tasks were decomposed into smaller subtasks, the use of sentinels would have to be redesigned throughout the whole program to enable fine-grained dependencies among these subtasks.

Finally, Figure 1c shows our proposed solution using `auto`. The precise semantics of `auto` will be described in Section IV. The `row` tasks have an output access on the pointers, `C[i][0;NT]`, allocated by the task, as well as an `auto` access to cover all the data accesses of their `matmul` subtasks, which are unknown at the time that the `row` tasks are created. The `none` access is an optimization to enable the `row` tasks to run concurrently, and it is described below. The `row` tasks create the `matmul` subtasks that will perform the matrix multiplication, and they finish without waiting for these subtasks to complete. Since the first `weakpotrf` task has a strong dependence on the pointer `C[0][0]`, it can execute and create its subtask as soon as the first `row` task finishes. When all the `matmul` tasks that calculate this block complete, the first `potrf` task can begin execution due to the dependency on `*C[0][0]`.

The `none` access on `row` is an optimization to allow concurrent creation of the full task dependency graph. It does not affect the fine-grained dependencies among the `matmul`, `potrf` and similar tasks (not shown) that do the majority of the work. A `none` access indicates that no accesses will need to be inferred in the region, beyond those explicitly expressed by the other task accesses. In particular, the `row` tasks do not create any subtasks that access any elements of C, except possibly `C[i][j]`. Without `none`, the runtime would have to consider this possibility, in which case the sequential ordering rules would require an ordering dependency between a subtask of one `row` task and a later `row` task. Since this situation remains a possibility until the earlier `row` task completes, the overall effect would be to serialize all the `row` tasks. The `none` clause says that this cannot happen, so all the `row` tasks can execute concurrently to build the dependency graph in parallel.

The version using the `auto` clause has several advantages. Firstly, the data accesses unify the specification of information needed to enforce task ordering, program data transfers and optimize data locality. Secondly, the annotations are clear and flexible, because they match the task's actual accesses. Thirdly, the task dependency graph can be constructed in parallel and well in advance of task execution, maintaining parallelism and providing maximum ability for the scheduler to optimize load balance and data locality.

B. *Productivity and incremental path for nested tasks*

Figure 2a shows an example program with nested tasks. Task `parent` creates a number of tasks, `child`, to do some of its work. In order to be properly nested according to the OmpSs-2 nesting rules, `parent` must itself have accesses, in at least the weak variant, covering all the accesses of its subtasks. This requires the multidependency on line 1, which is a burden on the programmer. It is redundant, as it duplicates information that

```
#pragma oss task depend(weakin: a[j][0;len[j]], i=0;N) \
         label("parent")
{
  // ...
  for(int j=0; j<N; j++) {
    #pragma oss task depend(inout: a[j][len[j]]) \
              label("child")
    { // Update a[j][0] ... a[j][len[j]]
      // ...
    }
  }
}
```

(a) Parent task requires a multi-dependency for its children

```
#pragma oss task depend(auto) label("parent")
{
  // ...
  for(int j=0; j<N; j++) {
    #pragma oss task depend(inout: a[j][len[j]]) \
              label("child")
    {// Update a[j][0] ... a[j][len[j]]
      // ...
    }
  }
}
```

(b) Parent task with `auto` clause

```
#pragma oss task depend(auto: a[0;M]) label("parent")
{
  // ...
  for(int j=0; j<N; j++) {
    #pragma oss task depend(inout: a[j][len[j]]) \
              label("child")
    { // Update a[j][0] ... a[j][len[j]]
      // ...
    }
  }
}
```

(c) Parent task with `auto` clause specifying a range

Fig. 2: Productivity and incremental development path for task nesting. The time consuming and error-prone parent task dependencies in (a) can be replaced by the `auto` clause in (b).

```
#pragma oss task depend(auto: <list>)
```

(a) Proposed OpenMP-style syntax

```
#pragma oss task auto
#pragma oss task auto(<region>)
```

(b) OmpSs-2-style syntax

Fig. 3: Proposed OpenMP and OmpSs-2 syntax for `auto`

	in	**out**	**inout**	**concurrent**	**commutative**	**reduction**	**none**	**auto**
in	in	inout	inout	inout	inout	invalid	in	in
out	-	out	inout	inout	inout	invalid	out	out
inout	-	-	inout	inout	inout	invalid	inout	inout
concurrent	-	-	-	conc.	comm.	invalid	conc.	conc.
commutative	-	-	-	-	comm.	invalid	comm.	comm.
reduction	-	-	-	-	-	reduction*	red.	red.
none	-	-	-	-	-	-	none	none
auto	-	-	-	-	-	-	-	auto

* Non-identical overlapping reductions on the same task are undefined.

TABLE I: Access upgrade rules, which define the combined effect of overlapping task accesses. This is the least restrictive access type that implies all ordering constraints and data transfers of the constituent accesses. An `auto` access is overridden by any specific access type, including `none`.

can be discovered by the runtime, and it is time consuming and error-prone to write these annotations for all the parent tasks.

Figure 2b shows the same example using the proposed `auto` dependency clause. It is clear that this approach allows a functional version to be obtained with much less effort. In terms of the ordering of the subtasks that do the majority of work, as well as data locality and data transfers, the behaviour is the same as that of Figure 2a. The only cost is a small amount of overhead, which may, if necessary, be incrementally reduced by refining the task accesses, guided by performance analysis. Figure 2c is a first step in performance optimization, where the programmer has declared that the subtasks of `parent` have unknown accesses, but they are all known to be in array `a[0;M]`. If a later task has a strong access on some other region of memory, then knowing that there can be no ordering constraint that would require it to execute after a subtask of `parent` allows the tasks to be executed concurrently.

IV. Programmer's model

A. Auto access type

The main extension to the programmer's model is the `auto` clause. It is in fact an access type, just like `in`, `inout`, `out` and so on, that by default covers the whole virtual address space from 1 to `SIZE_MAX` − 1 inclusive.[1] The proposed syntax is given in Figure 3, which shows OpenMP-style and OmpSs-2-style data access specifications. A task with an `auto` clause still requires strong data accesses for the data that is accessed directly by the task. But it does not in principle require any explicit (weak or strong) accesses on behalf of its subtasks.

1) Auto region: In a basic use, automatic aggregation of data accesses is done for any address in the whole virtual address space. But it may be known that automatic aggregation is needed only for certain regions, for instance unknown subtask accesses within a known array, memory allocations within a known memory pool or memory obtained from specially-mapped memory regions. In this case, one or more regions can be specified in the normal way for an access, i.e, as `depend(auto: addr[offset;size])`.

2) None access type: Additionally, it may be known that none of the descendent tasks of a task with the `auto` clause will access a certain region of memory. Non-accessed regions are specified using the `none` access type, which has similar syntax to the usual access types such as `in`, `inout` and `out`. A `none` data access is useful when performance analysis shows that a later task, that should execute concurrently, is actually serialized after the current task. This happens when the later task has a strong access on a region of memory and the runtime cannot know if the current task will create a subtask that accesses the same memory. This subtask would be ordered before the later task according to the sequential task ordering. Such serialization can sometimes be solved by narrowing the scope

[1]The access starts at 1 because accesses starting at `NULL` are ignored.

of the `auto` access by specifying an access region. But not always: an example that needs `none` is the `row` task in Figure 1c.

3) Upgrade rules: The upgrade table in Table I specifies how to resolve the situation when a task has multiple accesses covering the same memory region. For example a task that has both `depend(in: a)` (first row) and `depend(out: a)` (second column) is equivalent to one with the single access `depend(inout: a)`, since `inout` implies all necessary ordering constraints and data transfer requirements and there is no less restrictive data access type available. The combined access is strong if either access is strong. We extend to `auto` accesses by defining that `auto` is overridden by all other access types. So, for example the combined effect of overlapping `auto` and `weakin` accesses is `weakin`. A `none` access that overlaps any access type other than `auto` adds no additional ordering or data transfer requirements, so it has no effect. But `none` is a specific access type that overrides `auto`. The upgrade rules are commutative, so the combined effect of two accesses does not depend on the order in which they are written by the programmer. The table is therefore symmetric, and only the upper triangle is shown. The upgrade rules are also associative, so the order in which the upgrade rules are applied is not significant.

4) Fragmentation: It is valid in OmpSs-2, and therefore OmpSs-2@Cluster, for data accesses of the same or different tasks to partially overlap. The fragmented linear region dependencies, which is mandatory in the OmpSs-2 specification and the only supported dependency system for OmpSs-2@Cluster, will fragment (and unfragment) data accesses accordingly [20]. An `auto` access may initially cover a large part of the virtual address space, but it may be decomposed into subregions.

5) Inheritance of auto and none regions: If a task with the `auto` clause creates a subtask that in turn has one or more `auto` clauses, the subtask's clause will be restricted to cover at most the regions covered by parent accesses after the upgrade rules (other than `none`). This means that restrictions on the scope of analysis of a task and, by implication its descendents, can be provided once at the topmost level of the program, without having to duplicate this information throughout the codebase.

V. Implementation

A. Compiler

The only necessary change in the compiler is to add the new `auto` and `none` access types for task accesses. The compiler transformations for these access types are analogous to those for the existing `in`, `inout`, `out`, `concurrent` and `commutative` access types, the only difference being that an `auto` dependency is allowed to omit the access region. These two data access types are also added to the Nanos6 API. While sophisticated compiler analysis could be used to narrow the scope of the `auto` access to reduce overhead and avoid serialization, we have not found it necessary in our first implementation.

B. Runtime

1) Baseline implementation: A strong advantage of this proposal is that there is a simple baseline implementation. Three things are required. First, the runtime must respect the extended access upgrade rules in Table I. Secondly, it must remove all regions with `auto` access type that are not part of the parent accesses. It must also downgrade `auto` accesses inside a parent's `in` access to `weakin`. It must do this to conform to the programmer's model and to ensure proper nesting of data accesses. Thirdly, it must treat any remaining `auto` accesses in the same way as `weakinout`. This is clearly a valid implementation of the programming model. There is also no overhead for programs that do not have `auto` accesses. The overhead is already low for SMP (Section VII-B), but two optimizations are necessary for OmpSs-2@Cluster.

2) Optimizing non-accessed regions when offloading tasks: The job of a parent task with an `auto` access is typically to create the subtasks that will do the computations. Most of the `auto` access(es) will likely not be needed for subtask accesses. Such regions can be identified when the parent task completes, which is generally a long time before the data values are ready, and off of the critical path. These accesses can be recognised as those for which the runtime system has not registered any subtask in the *bottom map*, the map from addresses to the currently-last subtask (if one exists) that is used to build the dependency graph. The OmpSs-2@Cluster runtime already identifies such accesses and sends a message to the offloading node to prevent an unnecessary eager data send. When the region later becomes ready, there is normally a message to the remote node to indicate that it is ready, and another message back, passing this information to the next task. As an optimization, the offloading node skips this ping-pong, reducing the latency of the critical path.

3) Optimizing read-only regions when offloading tasks: In OmpSs-2@Cluster, if a task has an `in` access, then as soon as it is able to read the data, then this permission (read satisfiability) is immediately passed to the successor task, even if the task is offloaded, without going via the remote node. It is only necessary to inform back to the offloader when the access has completed. For an `auto` access that is read-only this ability is only passed back to a successor on a different node when the access is complete. The solution is to send a notification similar to the non-accessed notification of Section V-B2. On receipt of this notification the offloader sets a flag to indicate that read satisfiability can be immediately propagated to the next task.

VI. Methodology

A. 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 [21]. 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 Intel Omni-Path with a full-fat tree. GCC 7.2.0 was used to compile the benchmarks and the Nanos6@Cluster runtime. The runtime uses Intel MPI 2018.4, and the benchmarks use the BLAS functions provided by Intel MKL 2018.4.

Benchmark	Parameters	Description
SMP:		
`hypermatrix`	$N = 16384$ to 55296	Hypermatrix matrix multiplication followed by Cholesky decomposition
`matmul-smp`	$N = 4096$	Matrix multiply without BLAS using nested weak and strong tasks [12]
n-`body-smp`	$N = 262144$	$O(n^2)$ n-body code with two nested loops to determine the forces on the particles [12]
Distributed memory:		
`matvec`	$N = 65536$	Matrix–vector multiplication using nested weak and strong tasks [7]
`matmul`	$N = 32768$	Matrix-matrix multiplication using nested weak and strong tasks [7]
`jacobi`	$N = 65536$	Jacobi iteration with nested weak and strong tasks [7]
`cholesky`	$N = 65536$	Cholesky decomposition with nested weak and strong tasks, task for, memory reordering and priority [7]
n-`body`	$N = 1000000$	Barnes–Hut using nested weak tasks and strong taskloops

Fig. 4: Benchmarks, problem sizes and descriptions

B. Benchmarks

The benchmarks are listed in Table 4. On SMP, we use the matrix multiplication (`matmul-smp`) and n-body (n-`body-smp`) benchmarks from the OmpSs-2 examples [12] and the `hypermatrix` example from Section III-A. The block size for `hypermatrix` is 2048 by 2048 elements. Since the overhead of our implementation on SMP is low, we also include more challenging benchmarks using OmpSs-2@Cluster, executed on 2 processes per node (one per NUMA node) from 1 to 32 nodes (2 to 64 processes).

Four OmpSs-2@Cluster benchmarks are executed with as best as possible the same configurations as Aguilar et al. [7]. `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. `matmul` is a matrix–matrix multiplication with bigger tasks of $O(N^3)$ and a similar access pattern. `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. `cholesky` is a Cholesky decomposition 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 optimized version uses `task for` and memory reordering optimizations to reduce fragmentation and data transfers. All experiments use the empirically best blocksize or grainsize.

The n-body code [22] is an OmpSs-2@Cluster tasking implementation of Barnes–Hut [23]. The baseline implementation has a taskwait between construction of the tree and updating the particles. The optimized implementation uses an `auto` clause to replace the taskwait with a dependency.

VII. Results

A. Precise specification of data accesses without taskwaits

Figure 5 shows execution traces for the taskwait (Figure 5a) and `auto` (Figure 5b) variants of the `hypermatrix` example program from Section III-A. This trace is for a problem size of $N = 30720$, which corresponds to an upper triangular matrix of 3.5 GB, on all 48 cores of a single node. In Figure 5a, we see that all `matmul` tasks must complete before the Cholesky decomposition can start. This synchronization is due to the taskwait on line 16 of the program in Figure 1a. Since the number of `matmul` tasks is different for different blocks of the matrix multiplication, the loads on the cores are not perfectly balanced. The Cholesky decomposition involves the `trsm`, `gemm`, `syrk` tasks, as well as `potrf`, which is too small to see, and a number of weak tasks, which are also too small to see. In Figure 5b, we see the effect of the `auto` clause in correctly specifying the precise task dependencies without needing a taskwait or sentinel. Both traces use the same x-axis scale.

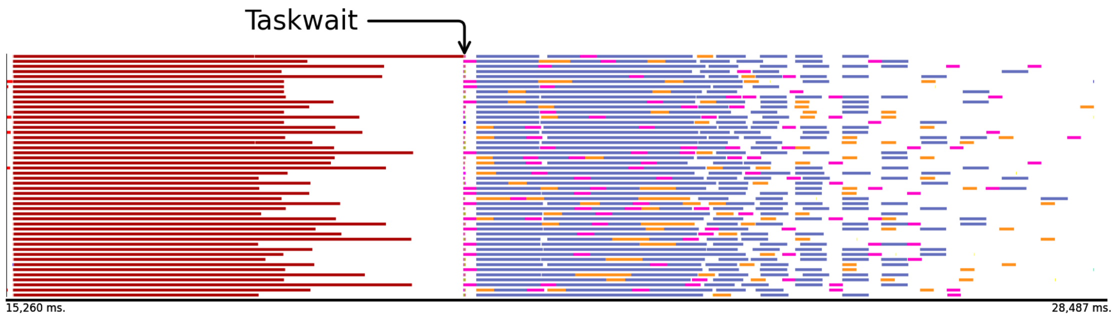
Figure 6a shows the GFLOP/s obtained for `hypermatrix`, as the problem size is varied between $N = 16384$ (1 GB) and $N = 55296$ (11.4 GB). All data points use all 48 cores of a single MareNostrum 4 node. They are the average of five executions and in all cases, standard deviation is $< 1\%$. For the larger problem sizes, there is a roughly 5% performance increase. While this improvement is not enormous, it does demonstrate the potential. It is limited by the critical path of the final part of the Cholesky decomposition and could likely be improved using a better task scheduling policy.

Figure 6b shows the performance of the `auto` clause for n-body, with strong scaling on 1 to 32 nodes with 1 process per node. Both versions build the tree, of unknown size, using a taskloop with a commutative dependency on the tree and another task calculates the forces and updates the particles using a taskloop. The taskloop is automatically distributed among the nodes, so it is important that the size of the tree is correct, to avoid copying too much data. The taskwait version needs a taskwait to obtain the size of the tree, whereas the `auto` version encloses the taskloop in a parent task with an `auto` data access. We see in Fig. 6b that the `auto` version has consistently higher throughput than the manual version with taskwaits, at 195,000 particles per second on 16 nodes, compared with 137,000 particles per second on 16 nodes for the manual version with taskwaits. This is a 1.4 times increase in throughput.

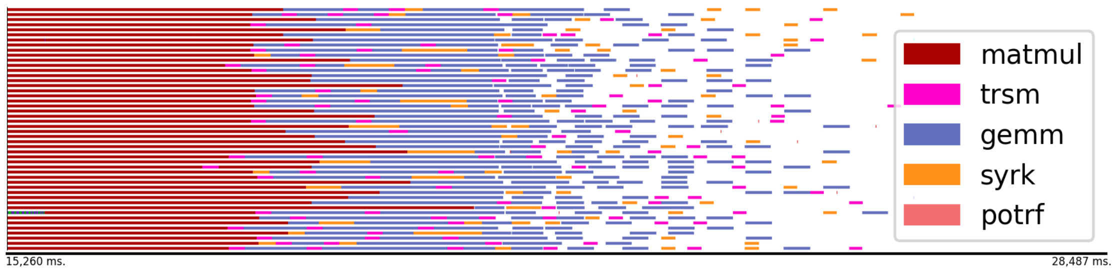
B. Automatic aggregation of subtask accesses

Figure 7 shows the performance of the `matmul-smp` and n-`body-smp` benchmarks on a single node. We see that across the whole range of block sizes, the simpler implementation using `auto` is always within 19.6% (`matmul-smp`) and 10% (n-`body-smp`) respectively of the original "manual" version.

Figure 8 shows the performance of the `auto` clause for the four benchmarks, `matvec`, `matmul`, `jacobi`, and cholesky, with strong scaling from 1 to 32 nodes. The x-axis is the number of nodes, each of which has two processes. The y-axis is the aggregate performance in GFLOP/s. The *manual* results are



(a) Using taskwait



(b) Using `auto` clause

Fig. 5: Extrae/Paraver trace of sparse hypermatrix multiplication followed by Cholesky decomposition. Both variants have precise data accesses on all tasks. Subfigure (a) has a taskwait and subfigure (b) uses the `auto` clause, which avoids the synchronization after the matrix multiplication. Both traces are shown on the same x-axis scale. The execution time of `potrf` is very short so these tasks are not visible in the traces.

for the *optimized* benchmarks in Aguilar et al. [7], which have complete and precise weak dependencies specified manually for each task. The *auto (unoptimized)* and *auto (optimized)* versions both have all of the weak accesses replaced by the `auto` clause, with the default region covering the full address space. The *auto (optimized)* version employs both optimizations described in Section V-B whereas the *auto (unoptimized)* version has them both disabled. We were able to fairly closely (within about 4%) reproduce the results in Aguilar et al. [7] for 1 to 16 nodes and extend to 32 nodes. In all cases, the `auto` (unoptimized) results are within 3% of manual on a single node, but the performance drops significantly on more than one node. The `auto` (optimized) results provide a reasonable level of scaling for a first functional version of the program with nested tasks. Figure 8a shows 3% performance degradation, for `matvec`, on up to 8 nodes, compared with the manual version. Similarly, Figure 8c shows 9% degradation on 8 nodes and 37% degradation on 16 nodes, again compared with the manual version. Figure 8d shows scaling only up to 4 nodes. Performance analysis using Extrae/Paraver showed excessive control message communication before one of the three offloaded tasks. Adding the true weak dependencies to that task enables scaling on up to 8 nodes with less than 36% degradation from the original "manual" version.

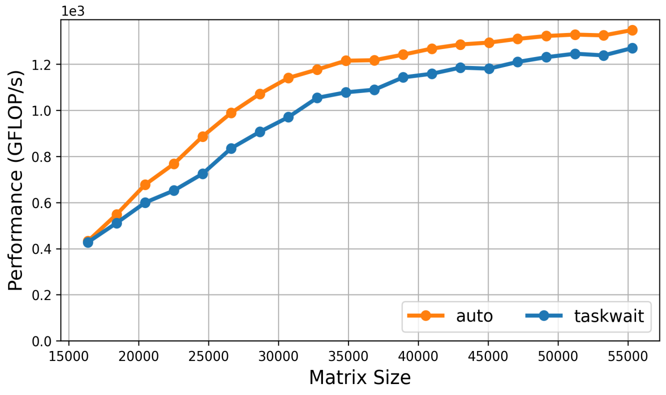
In summary, `auto` allows a functional version to be obtained that scales to a reasonable number of nodes, from 4 to 32 depending on the benchmark. It is therefore helps make an incremental step towards porting of nesting tasks.

VIII. Related work

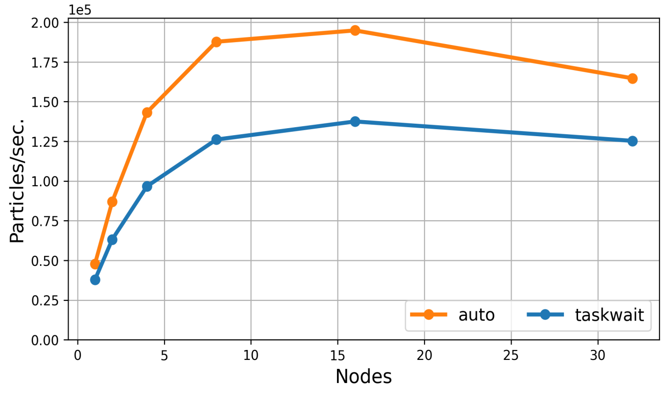
Compile-time setting of accesses: Many research efforts target automatic compile-time parallelization, e.g., Cetus [24], DawnCC [25], [26], AutoPar [27], Pluto [28] and TaskMiner [29]. These primarily target data parallelism, and only TaskMiner is specific to tasks. While there is some overlap with our work, the purpose is very different. These tools target conversion of sequential to parallel code. As they analyse the code at compile time, there is no runtime overhead, but they typically only support code that follows a particular structure. Our `auto` clause does not determine the strong accesses of the subtasks that do the work. Instead it aggregates information already known to the runtime so that fine-grained dependencies can be determined among subtasks that have been created by different concurrent parent tasks.

Run-time setting of accesses: OpenMP 5.1 [10] introduces `omp_all_memory`, which matches all accesses of previous sibling tasks. It is a convenient way to enforce a dependency that serializes with all prior tasks that have a specified access. Our proposal does not imply any ordering with respect to previous tasks. Rather it is a way to indicate that some of the weak accesses on behalf of subtasks have not been specified. We are aware of no related work that solves the same problem.

Automatic finding/verification of accesses: Tareador [30] uses an LLVM compiler stage to instrument all read–write instructions in a sequential application. This instrumentation

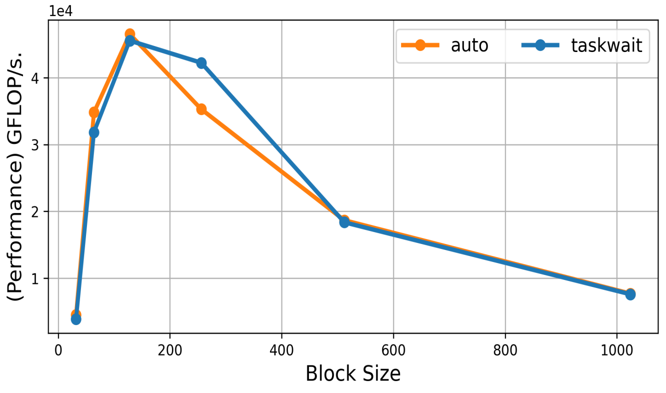


(a) Hypermatrix with different problem sizes on 1 node

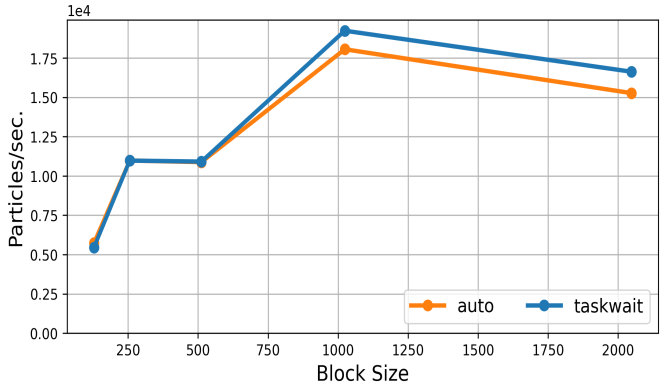


(b) Strong scaling of n-body on 1 to 32 nodes

Fig. 6: Performance of `auto` and taskwait versions of sparse hybermatrix on 1 node and n-body becnhamrks on 1 to 32



(a) `matmul-smp`



(b) n-`body-smp`

Fig. 7: Throughput of the `matmul-smp` and n-`body-smp` benchmarks on 1 node for the same problem size with different bloksizes. The version with `auto` to deduce all weak accesses is within 19.6% for `matmul-smp` and 10% for n-`body-smp` of the original `manual` version.

can determine the strong data accesses and help explore potential parallelism strategies. It has been successfully used for this purpose in undergraduate course on parallelism. StarSs-Check [31] is a Valgrind-based tool to verify correctness of the strong task accesses in a task-based program. Linter [32] is a run-time dynamic binary instrumentation tool that addresses the same problem. These tools all introduce enormous performance overhead of at least an order of magnitude. As demonstrated by our results, by concentrating on determining the weak accesses and assuming that the strong accesses are correct, our approach has dramatically smaller overhead.

IX. Conclusions

This paper proposes the `auto` clause, which indicates that the task annotations may be incomplete due to unspecified subtask memory accesses or memory allocation. The `auto` clause allows a task to be created before the data accesses of the task and its descendents are known. Existing approaches need to either block using a taskwait or substitute "fake" accesses known as sentinels. As there is no need to block, our approach enables concurrent task creation and execution to continue without interruption, maintaining parallelism and affording maximum freedom to the scheduler to optimize load balance and data locality. Since task annotations can match the actual data accesses, a single mechanism is used to control task ordering, program data transfers on distributed memory and optimize data locality. The `auto` clause also provides an incremental path to develop programs with nested tasks, because an initial functional implementation can be created without the time-consuming and error-prone specification of weak accesses on all parent tasks. We present a straightforward runtime implementation with a few key optimizations. We evaluate our the approach using a hypermatrix multiplication followed by Cholesky decomposition and a Barnes–Hut n-body application, which achieves a 1.4 times speedup on 32 nodes. We evaluate programmer productivity by replacing all weak accesses by `auto` on two SMP benchmarks and four OmpSs-2@Cluster benchmarks, and see a $< 4\%$ slowdown for three of the benchmarks on 8 nodes. All code is released open source [13] in the hope that future work will build on our results.

X. Acknowledgements

This research has received funding from the European Unions 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

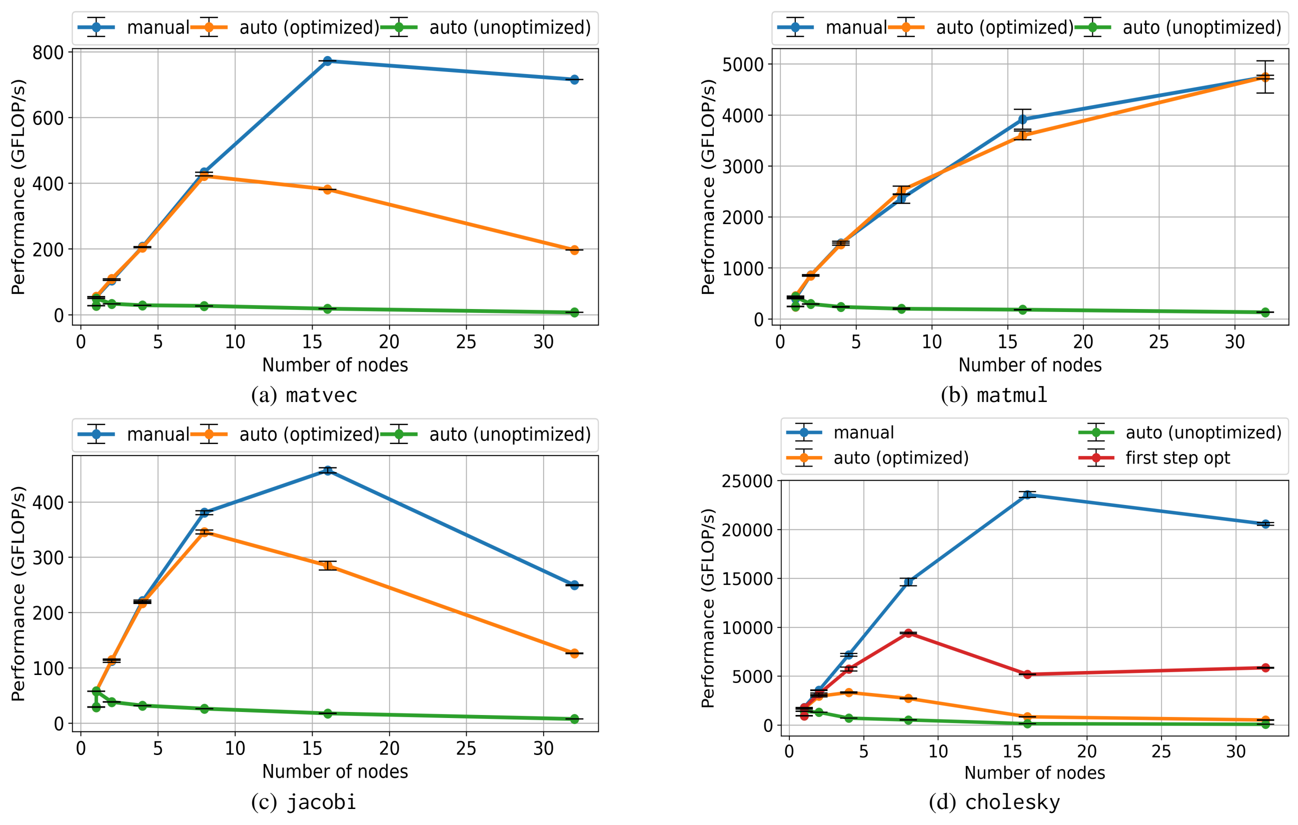


Fig. 8: Strong scaling benchmarkperformance for `matvec`, `matmul`, `jacobi` and `cholesky` benchmarks using OmpSs-2@Cluster on 1 to 32 nodes. The version with the `auto` clause and optimizations enabled in the runtime, achieves similar performance to the version with manual specification of dependencies on up to 4 to 32 nodes, depending on the benchmark.

Science and Innovation (contract PID2019-107255GB-C21/MCIN/AEI/10.13039/501100011033 and Ramon y Cajal fellowship RYC2018-025628-I/MCIN/AEI/10.13039/501100011033 and by "ESF Investing in your future"), as well as by the *Generalitat de Catalunya* (2017-SGR-1414).

References

[1] O. Aumage, P. Carpenter, and S. Benkner, "Task-based performance portability in HPC," Oct. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5549731

[2] R. Ferrer, J. Planas, P. Bellens, A. Duran, M. Gonzalez, X. Martorell, R. M. Badia, E. Ayguad, and J. Labarta, "Optimizing the exploitation of multicore processors and GPUs with OpenMP and OpenCL," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2010, pp. 215–229.

[3] J. Bosch, X. Tan, A. Filgueras, M. Vidal, M. Mateu, D. Jimnez-Gonzlez, C. lvarez, X. Martorell, E. Ayguad, and J. Labarta, "Application acceleration on FPGAs with OmpSs@FPGA," in *2018 International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 70–77.

[4] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguad, and J. Labarta, "Productive programming of GPU clusters with OmpSs," in *IEEE 26th International Parallel and Distributed Processing Symposium*, 5 2012.

[5] A. Zafari, E. Larsson, and M. Tillenius, "DuctTeip: An efficient programming model for distributed task-based parallel computing," *Parallel Computing*, 2019.

[6] 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.

[7] J. Aguilar Mena, O. Shaaban, V. Beltran, P. Carpenter, E. Ayguad, and J. Labarta, "OmpSs-2@Cluster: Distributed memory execution of nested OpenMP-style tasks," in *European Conference on Parallel Processing: Euro-Par 2022*, 2022.

[8] J. Klinkenberg, P. Samfass, M. Bader, C. Terboven, and M. Mller, "CHAMELEON: Reactive load balancing for hybrid MPI+OpenMP task-parallel applications," *Journal of Parallel and Distributed Computing*, vol. 138, 12 2019.

[9] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of OpenMP tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2008.

[10] OpenMP Architecture Review Board, "OpenMP Application Program Interface version 5.1," Nov. 2020. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[11] J. M. Perez, V. Beltran, J. Labarta, and E. Ayguad, "Improving the integration of task nesting and dependencies in OpenMP," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 809–818.

[12] Barcelona Supercomputing Center. (2022) OmpSs-2 examples. [Online]. Available: https://pm.bsc.es/gitlab/ompss-2/examples

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

[14] ——. (2021) OmpSs-2 specification. [Online]. Available: https://pm.bsc.es/ftp/ompss-2/doc/spec/

[15] ——. (2021) Influence in OpenMP - OmpSs-2 specification. [Online]. Available: https://pm.bsc.es/ftp/ompss-2/doc/spec/introduction/openmp.html

[16] D. Álvarez, K. Sala, M. Maroñas, A. Roca, and V. Beltran, *Advanced*

Synchronization Techniques for Task-Based Runtime Systems. New York, NY, USA: Association for Computing Machinery, 2021, p. 334347.

[17] Barcelona Supercomputing Center. (2021) Mercurium. [Online]. Available: https://pm.bsc.es/mcxx

[18] ——. (2021) Nanos6. [Online]. Available: https://github.com/bsc-pm/nanos6

[19] J. Aguilar Mena, O. Shaaban, V. Lopez, M. Garcia, P. Carpenter, E. Ayguad, and J. Labarta, "Transparent load balancing of MPI programs using OmpSs-2@Cluster and DLB," in *51st International Conference on Parallel Processing (ICPP)*, 2022.

[20] J. M. Péréz Cáncer, "A dependency-aware parallel programming model," 2015.

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

[22] P. Barkman, "Parallel Barnes–Hut algorithm," https://github.com/barkm/n-body, 2019.

[23] J. K. Salmon, "Parallel hierarchical n-body methods," Ph.D. dissertation, California Institute of Technology, 1991.

[24] H. Bae, D. Mustafa, J.-W. Lee, H. Lin, C. Dave, R. Eigenmann, S. P. Midkiff *et al.*, "The Cetus source-to-source compiler infrastructure: overview and evaluation," *International Journal of Parallel Programming*, vol. 41, no. 6, pp. 753–767, 2013.

[25] G. Souza Diniz Mendona, B. Campos Ferreira Guimares, P. R. Oliveira Alves, F. M. Quinto Pereira, M. M. Pereira, and G. Arajo, "Automatic insertion of copy annotation in data-parallel programs," in *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2016, pp. 34–41.

[26] G. Mendonça, B. Guimarães, P. Alves, M. Pereira, G. Araújo, and F. M. Q. Pereira, "DawnCC: automatic annotation for data parallelism and offloading," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[27] C. Liao, D. J. Quinlan, J. J. Willcock, and T. Panas, "Semantic-aware automatic parallelization of modern applications using high-level abstractions," *International Journal of Parallel Programming*, vol. 38, no. 5, pp. 361–378, 2010.

[28] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 101–113.

[29] P. Ramos, G. Souza, D. Soares, G. Araújo, and F. M. Q. Pereira, "Automatic annotation of tasks in structured code," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 1–13.

[30] E. Ayguadé, R. M. Badia, D. Jiménez, J. R. Herrero, J. Labarta, V. Subotic, and G. Utrera, "Tareador: a tool to unveil parallelization strategies at undergraduate level," in *Proceedings of the Workshop on Computer Architecture Education*, 2015, pp. 1–8.

[31] P. M. Carpenter, A. Ramirez, and E. Ayguad, "Starsscheck: A tool to find errors in task-based parallel programs," in *European Conference on Parallel Processing*. Springer, 2010, pp. 2–13.

[32] S. Economo, S. Royuela, E. Ayguadé, and V. Beltran, "A toolchain to verify the parallelization of OmpSs-2 applications," in *Euro-Par 2020: Parallel Processing*, M. Malawski and K. Rzadca, Eds. Cham: Springer International Publishing, 2020, pp. 18–33.