Specx: a C++ task-based runtime system for heterogeneous distributed architectures

Paul Cardosi, Bérenger Bramas

Specx: a C++ task-based runtime system for heterogeneous distributed architectures

Paul Cardosi, Bérenger Bramas
CAMUS Team, Inria Nancy - Grand Est, France
ICPS Team, ICube laboratory, France
Strasbourg University, France
ORCID 0000-0003-0281-9709
Berenger.Bramas@inria.fr

July 2022

Abstract

Parallelization is needed everywhere, from laptops and mobile phones to supercomputers. Among parallel programming models, task-based programming has demonstrated a powerful potential and is widely used in high-performance scientific computing. Not only does it allow for efficient parallelization across distributed heterogeneous computing nodes, but it also allows for elegant source code structuring by describing hardware-independent algorithms. In this paper, we present Specx, a task-based runtime system written in modern C++. Specx supports distributed heterogeneous computing by simultaneously exploiting CPUs and GPUs (CUDA/HIP) and incorporating communication into the task graph. We describe the specificities of Specx and demonstrate its potential by running parallel applications.

This document is a preliminary version of the publication on Specx, which does not include the benchmarks. We invite the readers to regularly check if a new version is available online.

1 Introduction

Modern computers are increasingly heterogeneous and structured hierarchically, both in terms of memory and parallelization. This is especially visible in the high-performance computing (HPC) environment, where clusters of computing nodes equipped with multi-core CPUs and several GPUs are becoming the norm. Programming applications for this type of architectures is challenging, and using them efficiently requires expertise.

The research community has proposed various runtime systems to help parallelize computational codes. These tools differ on many aspects, including the hardware they target, their ease of use, their performance, and their level of abstraction. Some runtime systems have demonstrated flexibility in their use, but they are designed for experts, such as StarPU [6]. Others provide a modern C++ interface, but they do not support as many features as what HPC applications need, such as Taskflow [31].

In our current study, we describe Specx (/'spɛks/) [1], a runtime system that has been designed with the objective of providing the features of advanced HPC runtime systems, while being easy to use and allowing developers to obtain modular and easy-to-maintain applications.

The contribution of our work can be summarized as follows:

- We describe the internal organization of Specx, a task-based runtime system written in modern C++.
- We present the key features needed to develop advanced HPC applications, such as scheduler customization, heterogeneous tasks, and dynamic worker teams.
- We show that Specx allows developers to write compact C++ code thanks to advanced meta-programming.
- Finally, we demonstrate the performance of Specx on several test cases.

The manuscript is organized as follows. We provide the prerequisites in Section 2 and the related work in Section 3. Then, we describe Specx in Section 4, before the performance study in Section 5.

2 Background

In this section, we briefly describe task-based parallelization and the challenges it faces when computing on heterogeneous architectures.

2.1 Task-based parallelization

Task-based parallelization is a programming model in which the application is decomposed into a set of tasks. It relies on the principle that an algorithm can be decomposed in interdependent operations, where the output of some tasks is the input of others. These tasks can be executed independently or in parallel, and they can be dynamically scheduled to different processing units while to ensure execution coherency. The result can be seen as a direct acyclic graph (DAG) of tasks, or simple graph of tasks, where each node is a task and each edge is a dependency. An execution of such a graph will start from the nodes that have no predecessor and continue inside the graph, ensuring that when a task starts, all its predecessors have completed. The granularity of the tasks, that is, the

[1] `https://gitlab.inria.fr/bramas/specx`

content in terms of computation, cannot be too fine-grained because the internal management of the graph implies an overhead that must be negligible to ensure good performance [48]. Therefore, it is usually the developer's responsibility to decide what a task should represent. The granularity is then a balance between the degree of parallelism and the runtime system overhead. For that reason, several researches are conducted to delegate partially or totally the runtime system to the hardware with the objective of relieving the worker threads, as in [18].

The dependencies between tasks can be described in various ways. One way is to have the user explicitly connect tasks together. For example, the user might call a function $connect(t_i, t_j)$ to connect tasks t_i and t_j. This approach requires the user to manage the coherency and to keep track of the dependencies between tasks, which can be error-prone and complicated between different stages of an application. TaskFlow uses this approach.

Another way is to inform the runtime system about the input/output of the tasks, and letting it taking care of the coherency. This approach is more convenient for the users, but there are many possibilities how this approach can be implemented. One approach is to use a mechanism like the C++ future to access the result of asynchronous operations. This approach allows the runtime system to track the dependencies between tasks and ensure that they are satisfied without having a view on the input/output. This approach is used by the ORWL runtime system [19].

An alternative is to use sequential task-flow (STF) [5], also called task-based data programming. In this approach, the users describes the tasks and tells what are the data input/output for each task. In general, a single thread creates the tasks and posts them in a runtime system, while informing about the access of each of them on the data. The runtime system is then able to generate the graph and guarantee that the parallel execution will have the absolute same result as a sequential one. This ends in a very compact code with few modifications required to add to an existing application by moving the complexity in the runtime system. The sequential order is used to set the dependencies caused by read after write, or write after read. This approach has a number of advantages, including:

- A sequential program can be transformed into a parallel equivalent very easily.
- The users do not have to manage the dependencies.
- The accesses can be more precise than read/write and specific properties can be set to the accesses, such as commutativity.
- The tasks can be mapped to a graph, allowing the runtime system to analyze the graph to predict the workload or memory transfers and takes clever decisions.

In our work, we use the STF model.

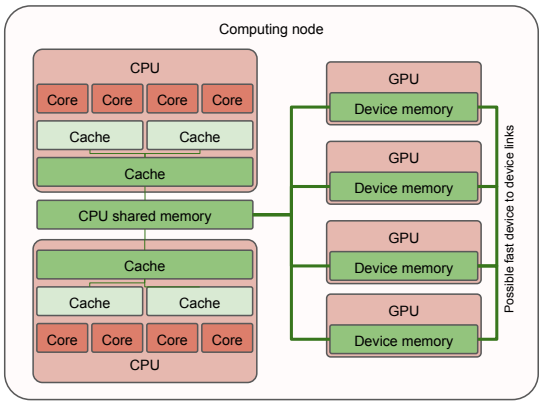


Figure 1: Simplified view of a heterogeneous computing node with 2 CPUs and 4 GPUs. Multiple nodes can be interconnected via network.

2.2 Computing on heterogeneous architectures

Heterogeneous computing nodes consist of at least two distinct types of processing units. The most common configuration includes a dual-socket CPU paired with one or several GPUs, each having separate memory nodes (Figure 1. However, similar principles apply to other types of processing units as well.

Traditionally, these nodes operate in a pattern where a single CPU thread manages data movement to the device's memory, initiates the computational kernel, and waits for its completion before transferring the data back if required. To assist programmers, vendors have introduced unified memory, a mechanism that creates the illusion of a shared memory space between CPUs and GPUs. However, due to potential unpredictability and lack of control, its use remains rare in High-Performance Computing (HPC). Meanwhile, this usage pattern leaves other CPU cores idle, which is untenable in HPC.

To enhance the utilization of processing units, this pattern can be expanded to incorporate multiple CPU threads sharing a single GPU, enabled by mechanisms like streams or queues. This arrangement allows full exploitation of the GPU in terms of computational capability and memory transfers. However, managing the device's memory becomes increasingly complex, as it becomes crucial to avoid redundant object copying and ensure memory capacity isn't exceeded.

Furthermore, this method introduces the key challenge of balancing heterogeneous computing. Specifically, determining the optimal number of CPU threads per GPU, figuring out how to best use idle CPU cores, and deciding which parts of applications are more suited for CPUs than GPUs. In essence, how can we optimally distribute work among all processing units?

Task-based runtime systems aim to resolve these issues. They predominantly manage data transfers between CPUs and GPUs, allocate specific CPU cores to control GPUs, separate these cores from others to allow for concurrent

executions, and schedule tasks across various types of processing units while considering workload and the most efficient processing unit type.

3 Related work

3.1 Task-based parallelization

The most common task-based programming pattern can be described as a tasks-and-wait scheme, where independent tasks are inserted into a runtime system and a synchronization point allows waiting for their completion. The task model from OpenMP 3 [40][8] and the task-based programming language Cilk [11] (later extended in Cilk++ [38] and Cilk Plus [32]) follow this idea. This remains a fork-join model because successive spawn phases of independent tasks (fork) must be explicitly synchronized (join) to ensure a correct execution. Therefore, it limits the scalability because of the waiting time and the imbalance between tasks. Of course, developers can increase the degree of parallelism by using multiple sources of tasks that they know are independent. However, such an implementation starts to become a manual management of the dependencies, which a modern task-based runtime system is intended to do.

This is why there now exist numerous different task-based runtime systems that support dependency management. The most popular ones are implementations of the OpenMP version 4 [41, 16] standard that defines the additional pragma keyword *depend* to inform the runtime system about the type of data accesses performed by the tasks. However, using pragmas, in general, is tedious when a task has hundreds of dependencies or when the number of dependencies are known at runtime. This can lead to ugly and error-prone code. In addition, as OpenMP is a standard, it is upgraded slowly to ensure backward compatibility. Moreover, the standard is weak in the sense that it does not impose any constraints on the implementation and complexity of the underlying algorithms. This can cause performance surprises for the users when they use different OpenMP runtime systems. In addition, OpenMP does not support distributed memory parallelization. Nonetheless, its portability, stability, and maturity make it a safe long-term choice.

StarPU [7] is a runtime system that was first designed to manage heterogeneous architectures. It is a C library, which means that users are constrained to use low-level programming and function pointers. However, it is extremely flexible and used by many HPC applications. It also supports distributed memory parallelization with three different approaches [3]. Each of these approaches uses a different description for the task graph, and the degree of information that the StarPU instances have on the complete graph is different. (there is one StarPU instance per computing node) The first approach is the most trivial, and it consists of declaring the complete graph on all computing nodes. This means that there is one thread that describes the graph in each StarPU instance. Each instance can analyze the graph without any communication, since it holds the complete graph. This can be used to create low-cost scheduling

strategies. For example, consider that all instances iterate over the task graph and have to decide on which computing node each task is executed. Thanks to the view on the complete graph, they can assign a task to a computing node while minimizing the communication, and all instances take the same decision without communicating. Moreover, the instances know where the data dependencies are located because they can track them while iterating on the graph. This can be used to post send/receive operations accordingly and manage the communication automatically. However, there is a clear disadvantage to this approach: the method cannot scale because its cost and overhead increase with the size of the task graph, independently of the number of computing nodes that will be used. In the second approach, each instance declares only a partial task graph that covers only the tasks it will compute. However, StarPU needs additional information to track the data movement and to connect the different partial task graphs that manage the communication. The first option consists in requesting explicit communication calls (similar to the MPI) that connect the tasks between the instances. In the second option, each instance inserts the tasks that will be computed by others and which are at the frontier of its partial graph. These two approaches remove abstraction because the developers manually split the task graph and have to manage the boundaries of the partial graph. Moreover, each instance has only a partial view, making analysis and scheduling difficult. Specx uses a similar approach.

PaRSEC [12, 21] is a runtime system based on the parametrized task graph (PTG) model [20]. It has been demonstrated to be effective in various scientific applications. The PTG is a domain-specific language (DSL) that captures a static, algebraic description of a task graph that can be expanded efficiently at runtime. This allows PaRSEC to manage large graphs without fully instantiating them. This approach works well on affine loops thanks to polyhedral analysis. The analysis of the data-flow of a task instance is constant in time, and the representation of the graph is constant in space. This makes the PTG a very efficient way to represent task graphs. However, the PTG is not as expressive as other task graph models. It is difficult to use the PTG to represent applications with irregular or sparse algorithmic or data access patterns. Despite this limitation, the PTG has been shown to be effective in a wide variety of scientific applications. It is a powerful tool for parallelizing applications that can be expressed in terms of affine loops. It is theoretically possible to write PTGs for highly dynamic applications, but this would imply an unbounded amount of time building and traversing dynamic meta-data in memory. However, the PTG is impractical for implementing applications with irregular or sparse algorithmic or data access patterns, where the logic is difficult to express with linear equations. The PTG graph representation is highly efficient, but the expressiveness of the model is limited. Internally, the representation allows collapsing a task graph in two dimensions, i.e., time and parallelism [45], which permits several optimizations. In distributed-memory, the different Parsec instances all hold an algebraic representation of the complete graph. Parsec uses advanced mechanisms to allow scheduling the tasks efficiently using heuristics and potential input from the users.

Charm++ [34, 35, 15] is an object-oriented parallel programming framework that relies on a partitioned global address space (PGAS) and that supports the concept of graphs of actors. It includes the parallelism by design with a migratable-objects programming model, and it supports task-based execution. The actors (called *chares*) interact with each other using invocations of asynchronous methods. However, with Charm++, there is no notion of tasks as we aim to use. Instead, tasks are objects that communicate by exchanging messages. Charm++ schedules the chares on processors and provides object migration and a load balancing mechanism. PGAS allows accessing data independent of their actual location, which is the inverse of what the task-based method intends to offer. A task is a piece of work that should not include any logic or communication. This approach forbids many optimizations and mechanisms that task graphs support [49].

HPX [33] is an open-source implementation of the ParalleX execution model. Its implementation aims to respect the C++ standard, which is an asset for portability and compliance with existing C++ source code. In HPX, tasks request access to data by calling an accessor function (get/wait). The threads provide the parallelism description, which is tied to the order and type of data accesses.

OmpSs [22, 42] uses the insert-task programming model with pragmas similar to OpenMP through the Nanos++ runtime to manage tasks. When running in distributed memory, it follows a master-slave model, which may suffer from scalability issues as the number of available resources or the problem size increase.

XKaapi [27] is a runtime system that can be used with standard C++ or with specific annotations, but it requires a specific compiler. Legion [9] is a data-centric programming language that allows for parallelization with a task-based approach. SuperGlue [52] is a lightweight C++ task-based runtime system. It manages the dependencies between tasks using a data version pattern. X10 [17] is a programming model and a language that relies on PGAS, too, and hence has similar properties as Charm++. Intel Threading Building Blocks [10] (ITBB) is an industrial runtime system provided as a C++ library. It is designed for multicore parallelization or in conjunction with oneAPI, but it follows a fork-join parallelization pattern.

Regarding distributed parallelization, most runtime systems can be used with MPI [44]. The developers implement a code that alternate between calls to the runtime systems and post of MPI communications. When supported by the runtime system, the data movements between CPUs/GPUs and in-node load balancing are delegated to the runtime system. More advanced methods have been elaborated, and they entirely delegate the communications to the runtime system [55, 54, 36, 30, 25], like Parsec, StarPU [2], Legion, Charm++, TaskTorrent [14], and HPX.

Most of these tools support a core aspect of a task-based runtime system, including the creation of a task graph (although the implementation may vary) where tasks can read or write data. However, scheduling is an important factor in the performance [4], and few of these runtime systems propose a way to create

a scheduler easily without having to modify the code. Moreover, specific features offer mechanisms to increase the degree of parallelism. For instance, some runtime systems permit the specification of whether data access is commutative, implying that tasks write data without any particular order. This kind of advanced functions can significantly impact performance [1]. They differ whether the task graphs are statically or dynamically generated, how the generation is performed, which in-memory representation is used, which parallelization levels are supported, and many other features [51, 50, 28, 29].

3.2 Speculative execution

Speculative execution is an approach that can increase the degree of parallelism. It has been widely used on hardware and is an ongoing research topic in regard to software [23, 37, 39]. The key idea of speculative execution is to utilize idle components to execute operations in advance, which includes the risk of performing actions that may later be invalidated. The prominent approach is to parallelize an application, and to detect at runtime if race conditions or accesses with invalid orders which violate dependencies happen. The detection of invalid speculative execution can be expensive, and as a result, some research is intended to design hardware modules for assistance [47, 43]. However, these low-level strategies are unsuitable for massively parallelized applications and impose the need for either detecting the code parts suitable for speculation or relying on explicit assistance from developers.

In a previous study, we have shown that characterizing accesses as 'maybe-write' instead of 'write' allows us to increase the degree of parallelism thanks to speculative execution in the task-based paradigm [13]. This novel kind of uncertain data access (UDA) can be used when it is uncertain at task insertion time whether the tasks will modify some data or not. Similar to the 'commutative write', developers simply provide additional information to the runtime system, enabling it to set up a strategy by modifying the graph of tasks on the fly. This also makes it possible to delay some decisions from the implementation time to the execution time, where valuable information about the ongoing execution is available. We have implemented this mechanism in our task-based runtime system Specx (originally called SPETABARU) and conducted an evaluation on Monte Carlo simulations, which demonstrated significant speedups. We are currently developing a new model [46].

On different fields, speculation has also been used in a tasking framework for adaptive speculative parallel mesh generation [53] and for resource allocation in parallel trajectory splicing [26].

4 Specx's features, design and implementation

4.1 Task graph description

In Specx, we dissociated the task-graph from the so-called computing engine that contains the workers. Therefore, the user has to instantiate a task-graph and select among two types, one with speculative execution capability and one without, which allows the removal of the overhead of the speculative execution management when no UDAs will be used. We provide an example in Code 1.

```
// Create a task graph
SpTaskGraph<SpSpeculativeModel::SP_NO_SPEC> tg;
// Legacy version, create a runtime (a compute engine + a task graph)
SpRuntime runtime(SpUtils::DefaultNumThreads());
```

Code 1: Specx example - creation of a task graph.

Task Insertion Specx follows the STF model: a single thread inserts the task in the runtime system (task-graph object) and tells which variables will be written or read. Additionally, the user can pass a priority that the scheduler is free to use when making decisions. The core part of the task consists of a callable object with the operator *()*, which allows for the use of C++ lambda functions. The data access modes that Specx currently supports are:

- SpRead: the given dependency will only be read by the task. As such, the parameter given to the task function must be *const*.
- SpWrite: the given dependency will be read or write by the task.
- SpCommutativeWrite: the given dependency will be read or write by the task but the order of execution of all the *SpCommutativeWrite* inserted jointly is not important.
- SpMaybeWrite: the given dependency might be read and/or write by the task. Possible speculative execution patterns can be applied.
- SpAtomicWrite: the given dependency will be read or write by the task, but the user will protect the access by its own mechanisms (using mutual exclusion, for example). The runtime system manages this access very similarly to a read access (multiple *SpAtomicWrite* can be done concurrently, but the runtime system has to take care of the read-after-write, write-after-read coherency).

When a dependency X is passed, the runtime dereferences X to get its address, and this is what will be used as the dependency. An important point when using task-based programming is that it is the user's responsibility to ensure that the objects will not be destroyed before all tasks that use them are completed. We provide an example in Code 2.

```
const int initVal = 1;
int writeVal = 0;
// Create a task with lambda function
tg.task(SpRead(initVal), SpWrite(writeVal),
             [](const int& initValParam, int& writeValParam){
    writeValParam += initValParam;
});
```

Code 2: Specx example - creation of a task for CPU.

Dependencies on a Subset of Objects A critical drawback of OpenMP is the rigidity of the dependency declaration. Indeed, the number of dependencies of a task has to be set at compile time. For example, if we use a vector of objects and want to declare a dependency on all or some elements and not on the vector, we cannot do it in OpenMP if the size of the vector is not known when we write the code because we have to write one pragma *depend* statement per dependency.

To solve this issue, in Specx, we can declare the dependencies on a set of objects using the following mechanisms: *SpReadArray(<XTy> x, <ViewTy> view)*, *SpWriteArray(<XTy> x, <ViewTy> view)*, *SpMaybeWriteArray(<XTy> x, <ViewTy> view)*, *SpCommutativeWriteArray(<XTy> x, <ViewTy> view)*, *SpAtomicWriteArray(<XTy> x, <ViewTy> view)*, where *x* should be a pointer to a contiguous buffer (or any container that support the *[]* operator), and *view* should be an object representing the collection of specific indices of the container elements that are affected by the dependency. *view* should be iterable (in the sense of "stl iterable").

With this mechanism, Specx can iterate over the elements and apply the dependencies on the selected ones. We provide an example in Code 3.

```
std::vector<int> vec = ...;
// Access all the elements in the SpArrayView
tg.task(SpPriority(1), SpWriteArray(vec.data(),SpArrayView(vec.size())),
            [](SpArrayAccessor<int>& vecView){
    ...
});
```

Code 3: Specx example - use array of dependencies.

Task Viewer Inserting a task in the task-graph returns a task view object, which allows accessing some attributes of the real task object. For instance, it allows setting the name of the task, waiting for the task completion, or getting the value produced by the task (in case the task returns a value). Unfortunately, there is a pitfall with the current design, which is the fact that accessing the task through the viewer can potentially be done after the task has been computed. For instance, we cannot use the tasks' names in the scheduler because they might be set after the tasks are computed. We provide an example in Code 4.

```
auto taskViewer = runtime.task(SpRead(initVal), SpWrite(writeVal),
             [](const int& initValParam, int& writeValParam) -> bool {
    writeValParam += initValParam;
    return true;
});
taskViewer.setName("The name of the task");
taskViewer.wait(); // Wait for this single task
taskViewer.getValue(); // Get the value (when the task is over)
```

Code 4: Specx example - task viewer.

4.2 Teams of Workers and Compute Engines

Within Specx, a team of workers constitutes a collection of workers that can be assigned to computational engines. In the current implementation, each worker is associated with a CPU thread that continuously retrieves tasks from the scheduler and handles them. If the worker is CPU-based, the task is directly executed by the CPU thread. Conversely, in the case of a GPU worker, the CPU thread manages the data movement between memory nodes and call the device kernel.

A compute engine necessitates a team of workers and may be responsible for several task-graphs. Currently, it is not possible to change the compute engine assigned to a task-graph, but it is possible to shift workers among different compute engines. This feature provides the ability to dynamically adjust the capabilities of the compute engine during execution and design advanced strategies to adapt to the workload of the graphs.

Given that dependencies among task-graphs are not shared, the insertion of tasks and their dependencies into a task-graph does not affect others. This allows for the creation of recursive parallelism, in which a task-graph is created within a task. Such a task-graph could potentially be attached to the same compute engine as the parent task. This approach could help mitigate the overhead associated with the creation of a large set of tasks by organizing them into sub-task-graphs. We provide an example in Code 5.

```
SpTaskGraph<SpSpeculativeModel::SP_NO_SPEC> tg;
// Create the compute engine
SpComputeEngine ce(SpWorkerTeamBuilder::TeamOfCpuWorkers(NbThreads));
// OR
SpComputeEngine ce(SpWorkerTeamBuilder::TeamOfCpuCudaWorkers());
// Tells which compute engine will manage the graph
tg.computeOn(ce);
```

Code 5: Specx example - creation of a compute engine.

4.3 Tasks for Heterogeneous Hardware

Specx relies on the same principles as StarPU in supporting heterogeneous hardware, i.e., we have distinct workers for each type of processing unit, and each

task can operate on CPUs, GPUs, or both. Specifically, at task insertion, we require a unique callable object for each processing unit type capable of executing the task. During execution, the scheduler determines where the task will be executed. This represents a critical challenge in task-based computing on heterogeneous systems.

Regarding the interface, the primary challenge is the movement of data between memory nodes. More specifically, we strive to exploit C++ and use an abstraction mechanism to facilitate object movement. Consequently, we have determined that objects passed to tasks should comply to one of the following rules: 1) the object is trivially copyable [2]; 2) the object is a std::vector of trivially copyable objects; 3) the object's class implements specific methods that the runtime system will call.

In the last case, the object's class must have as a class attribute a data type called *DataDescriptor* and three methods:

- *memmovNeededSize*: Invoking this method on the object should yield the required size of the memory to be allocated on the device for copying the object.

- *memmovHostToDevice*: This method is called to transfer the object to the device. The method receives a mover class (with a copy-to-device method) and the address of a memory block of the size determined by *memmovNeededSize* as parameters. The method may return a *DataDescriptor* object, which will later be passed to *memmovDeviceToHost* and to the task utilizing the object.

- *memmovDeviceToHost*: This method is invoked to move the data back from the GPU to the object. The method receives a mover class (with a copy-from-device method), the address of a GPU memory block, and an optional *DataDescriptor* object as parameters.

From a programming perspective, we require the users to determine how the data should be moved as they have the knowledge to do so. For example, consider an object on a CPU being a binary tree where each node is a separate memory block. It would be inefficient to allocate and copy each node. Consequently, we ask the users to estimate the needed memory block size, and we perform a single allocation. Then it is the users' responsibility to mirror the tree on the GPU using the block we allocated, and to implement the task such that it can use this mirror version. This design may change in the future as we continue to apply Specx to existing applications.

Currently, we employ the Least Recently Used (LRU) policy to determine which memory blocks should be evicted from the devices when they are full. Concretely, this implies that when a task is about to be computed on the device, the worker's thread will iterate over the dependencies and copy them onto the GPU's memory using a stream/queue. If an object already has an up-to-date version on the device, the copy will be skipped, and if there is not enough free

[2] `https://en.cppreference.com/w/cpp/types/is_trivially_copyable`

memory, older blocks may be evicted. As a result, at the end of a simulation, the up-to-date versions of the objects might be on the GPUs, necessitating their transfer back to the CPUs if required. At present, this can be accomplished by inserting empty CPU tasks that use these objects.

By default, worker teams align with hardware configurations, i.e., they will contain GPU workers for each available GPU. Therefore, if users are not careful and only need one type of processing unit for their tasks, the hardware will be underutilized as some workers will remain idle. We provide an example in Code 6.

```
class Matrix{
    int nbRows;
    int nbCols;
    std::vector<double> values;
public:
    // What to allocate on the device
    std::size_t memmovNeededSize() const{
        return sizeof(double)*nbRows*nbCols;
    }

    // Copy to the device (size == memmovNeededSize())
    template <class DeviceMemmov>
    auto memmovHostToDevice(DeviceMemmov& mover, void* devicePtr, ...
     std::size_t size){
        double* doubleDevicePtr = reinterpret_cast<double*>(devicePtr);
        mover.copyHostToDevice(doubleDevicePtr, values.data(), ...
     nbRows*nbCols*sizeof(double));
        return DataDescr{rowOffset, colOffset, nbRows, nbCols};
    }

    // Copy to the CPU
    template <class DeviceMemmov>
    void memmovDeviceToHost(DeviceMemmov& mover, void* devicePtr, ...
     std::size_t size, const DataDescr& /*inDataDescr*/){
        double* doubleDevicePtr = reinterpret_cast<double*>(devicePtr);
        mover.copyDeviceToHost(values.data(), doubleDevicePtr,  ...
     nbRows*nbCols*sizeof(double));
    }
};

// ....
Matrix matrix;

tg.task(SpPriority(1), SpWrite(matrix),
    SpCpu([](Matrix& matrix){
        // ...
    })
#ifdef SPECX_COMPILE_WITH_CUDA
  , SpCuda([](SpDeviceDataView<Matrix> matrix) {
        // ...
    })
#endif
).setTaskName(std::string("My operation"); // Set the name of the task
```

Code 6: Specx example - creation of a task for CPU/GPU.

4.4 Mixing Communication and Tasks

In the context of distributed memory parallelization, Specx provides the capability to mix send/receive operations (MPI) and computational tasks. Putting MPI communications directly inside tasks will fail due to the potential concurrent accesses to the communication library (which is not universally supported by MPI libraries) and the risk of having workers waiting inside tasks for communication completion, leading to deadlocks if tasks sending data on one node do not coincide with tasks receiving data on another. Therefore, to avoid having the workers dealing with communication, our solution is to use a dedicated background thread that manages all the MPI calls.

In this approach, a *send* operation is transformed into a communication task that does a write access on the data, with execution will be done by the background thread. Similarly, a *receive* operation becomes a communication task that performs a read access, and is also managed by the background thread. Once a communication task is ready, the background thread executes the corresponding non-blocking MPI calls, receiving an MPI request in return. This request is stored in a list, which the background thread aims to complete by calling the MPI *test-any* function. When a request is fulfilled, the background thread releases the dependencies of the associated communication task, thereby ensuring progression of the task-graph execution. In this way, the progression is done as early as possible.

In order to send/receive C++ objects using MPI in a single communication (although we perform two - one for the size and one for the data), we need a way to store the object into a single array. To achieve this, the object must comply with one of the following rules:

- It should be trivially copyable;
- It should provide access to a pointer of the array to be sent (or received). For example, if a class has virtual methods, it will not be trivially copyable. However, if the class's only attribute is a vector of integers, sending the object is equivalent to sending the vector's data.
- It should support our serialization/deserialization methods. Here, we allow the object to serialize itself using our utility serializer class, yielding a single array suitable for communication. Upon receipt, the buffer can be deserialized to recreate the object. This method offers the more flexibility, but is also the less efficient.

Specx also supports MPI broadcast as part of MPI global communication functions. Currently, users must ensure that all Specx instances perform the same broadcasts and in the same order.

As a side note, MPI communications are incompatible with the speculative execution capabilities of Specx due to the potential creation of extra tasks and instantiation of diverse execution paths. We provide an example in Code 7.

```
class Matrix{
    // ...
    Matrix(SpDeserializer &deserializer)
        : nbRows(deserializer.restore<decltype(nbRows)>("nbRows")),
          nbCols(deserializer.restore<decltype(nbCols)>("nbCols")),
          values(deserializer.restore<decltype(values)>("values")){
    }

    void serialize(SpSerializer &serializer) const {
        serializer.append(nbRows, "nbRows");
        serializer.append(nbCols, "nbCols");
        serializer.append(values, "values");
    }
};

// ....
tg.mpiRecv(matrix, srcRank, tag);
// ....
tg.mpiSend(matrix, destRank, tag);
```

Code 7: Specx example - sending/receiving a matrix object (using the serializer method).

4.5 Scheduling

We designed the scheduler module following the implementation approach used in StarPU, with a scheduler providing two key functions: *push* and *pop*. When a task becomes ready (i.e., its predecessors are finished), it is pushed into the scheduler. Conversely, when a worker becomes available, it calls the pop function on the scheduler, which may potentially return void if there are no tasks compatible with its processing unit type, or if the scheduler makes such a decision. As such, the scheduler plays a crucial role as it manages task distribution and the order of task execution.

At present, Specx utilizes a simple First-In-First-Out (FIFO) scheduler, but we plan to introduce more sophisticated schedulers [24] in the near future. Anyway, it is straightforward to provide a new scheduler, and users have the flexibility to implement a custom scheduler specifically designed for their application. This can be accomplished by creating a new class that inherits from our abstract scheduler interface.

4.6 Speculative execution

Specx supports task-based speculative execution, which is an ongoing research problem. We currently support two speculative models applicable when certain data accesses are flagged as *maybe-write*. In these scenarios, the runtime system may duplicate some data objects and tasks to enable potential speculative work, subsequently performing a rollback if the uncertain tasks actually modified the data.

4.7 Internal implementation

In this section, we delve into the finer details of Specx's implementation. When a task is inserted, the callable's prototype should match with the dependency types. Hence, read parameters are passed as *const*. For CPU callables, parameters should be of the type references to the object types passed as arguments. Using a value instead of a reference will simply result in a copy, which is typically not the intended outcome. Indeed, if values are required but are not significant as dependencies, it is more appropriate to pass them as captures in the lambda.

When an object is passed to a task, the runtime system dereferences it to obtain its address. This address is utilized as a dependency value and also as a key in an unordered hashmap that matches pointers to data handles. A data handle is a class that contains all the necessary information the runtime system requires concerning a dependency. For instance, the data handle includes the list of dependencies applied to the associated object. This allows progression in the list and subsequent release of dependencies. In terms of implementation, we do not construct a graph; instead, we have one data handle per address that has been used as a dependency, and the data handles' dependency lists contain pointers to the tasks that use the related objects. Consequently, when a task is finished, we increment a counter on the dependency list and access the next tasks. Doing so, we then examine whether the task now pointed to by the updated counter is ready, and we push the task into the scheduler if that is the case. The data handle also possesses a mutual exclusion object that enables its locking for modification. When several data handles need to be locked, we sort them based on their address, ensuring deadlock prevention.

The commutative (*SpCommutativeWrite*) dependency is managed differently because the related tasks' order is not static. Said differently, when the next tasks use the commutative access, we do not know which one should be executed. As such, we cannot merely point to the first task in the dependency list and stop our inspection if it's not ready, as the following tasks that also have commutative access to the dependency might be ready. This necessitates a check on all the tasks performing a commutative access at the same time point. However, several threads that completed a task might do so simultaneously, requiring us to use a mutual exclusion to protect all the commutative dependencies. In other words, using commutative dependencies implies the use of global mutual exclusion.

We use C++ meta-programming massively, such as testing if an object is trivially copyable or supports serialization methods, for example. We also utilize the inheritance/interface pattern and the template method design pattern. For instance, this enables us to have a task class containing the callable type, and thus the types of all parameters and arguments. We can then carry out meta-programming tests on the arguments to ensure compliance with specific rules, etc.

Finally, as we use a hashmap to store the data dependency objects' information, using their address as keys, it's currently undefined behavior to have objects of different types but stored at the same address. This primarily occurs

when an object of type x is freed, and subsequently, an object of type y is allocated using the same memory block. This is because the data handle class uses a data copier through an interface, but the copier is actually templatized over the dependency object type.

4.8 Visualization

Profiling and optimizing task-based applications are crucial to achieve high performance. The main information are:

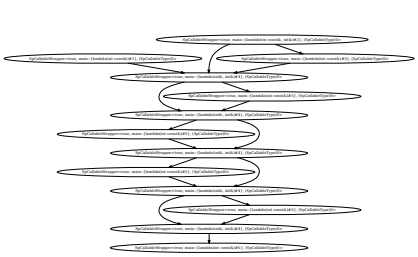
- Degree of parallelism: This represents how many tasks can be executed in parallel. The task graph can be used to evaluate if the degree of parallelism is sufficient to utilize all the processing units fully. Furthermore, during the execution, the number of ready tasks over time can also be analyzed.
- Task granularity: The task granularity can impact the degree of parallelism. An examination of an execution trace can help determine if the granularity is too small. If so, the overhead of task management and/or data displacement may be too large compared to the task durations, thereby negatively affecting performance. Conversely, if the tasks are too large, the degree of parallelism can be too small, and the end of the execution can be penalized with too few large tasks to compute.
- Scheduling choices for task distribution: If a slow worker is selected mistakenly (a worker that can compute a task but is not efficient to do so), it can reduce performance. It could be faster to wait and assign the tasks to a quicker worker, but this depends on the scheduler.
- Scheduling choices for task order: The degree of parallelism (and sometimes the availability of suitable tasks for all workers) can be influenced by the order of task execution, that is, the choice among the ready tasks.

An good situation, but not necessarily optimal, is when no workers have been idle, and the tasks have been assigned to the processing units that can execute them most efficiently.

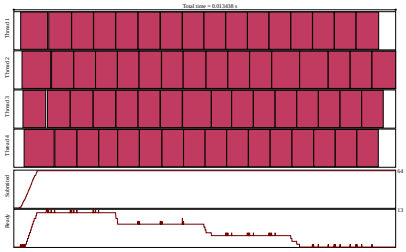
To facilitate the profiling, Specx provides features to export the task-graph and execution trace. The task-graph is generated in the *dot* format.[3] For the execution trace, a SVG file[4] is exported that can be opened with any modern internet browser. The execution trace also indicates the number of tasks available during the execution. In the next release, we plan to export metrics that will provide concise but meaningful numbers on execution quality, such as the idle time.

A graph and an execution trace are provided in Figure **??**, and the corresponding calls are given in Code .

[3] `https://gitlab.com/graphviz/graphviz`
[4] `https://www.w3.org/Graphics/SVG/`



(a) Task graph of the *daggraph* example, generated from the dot file.



(b) Execution trace of the GEMM test case using 4 threads, a matrix of size 512 and blocks of size 128.

Figure 2: Example of graphs and execution trace exported after a run.

```
// Export the graph
tg.generateDot("/tmp/graph.dot");
// Export the execution trace (false means: hide the dependencies)
tg.generateTrace("/tmp/gemm-simu.svg", false);
```

Code 8: Specx example - export of the task graph and the execution trace.

5 Performance and usability study

5.1 Configuration and test cases

We assess our method on two configurations:

- Intel-AVX512: it is a 2 × 18-core Cascade Lake Intel Xeon Gold 6240 at 2.6 GHz with AVX-512 (Advanced Vector 512-bit, Foundation, Conflict Detection, Byte and Word, Doubleword and Quadword Instructions, and Vector Length). The main memory consists of 190 GB DRAM memory arranged in two NUMA nodes. Each CPU has 18 cores with 32KB private L1 cache, 1024KB private L2 cache, and 25MB shared L3 cache. We use the GNU compiler 11.2.0 and the MKL 2022.0.2.

5.2 Results

Overhead In this section, we discuss the engine overhead that we evaluate with the following pattern. We create a runtime system with T CPU workers and T distinct data objects. Then, we insert $T \times N$ tasks, with each task accessing one of the data object. Consequently, the task graph we generate is actually composed of T independent sub-graphs. Inside each task, the worker that execute it simply waits until for a given duration D. As a result, the final execution time is given $N \times (D + O)$, where O is an overhead of picking a task

from the runtime. Also, we can measure the time it takes to insert the $T \times N$ tasks to obtain an insertion cost I.

We provide the result in Figure 3 for 1 to 20 dependencies. As expected, the overhead of using commutative write is significant compared to a normal write. The insertion cost is also higher when the tasks duration is smaller ($D = 10^{-5}$). The reason is that as the tasks are smaller, the worker query to the runtime system more often, which can compete with the insertion of ready tasks by the master thread and create contention. The cost also increase slightly as the number of dependencies in the tasks increases. Finally, the overhead per tasks is stable as the number of dependencies per tasks increases for the write access. However, for the commutative access, the overhead increases with the number of dependencies.

Application This part is not ready yet.

6 Conclusion

We presented Specx, a task-based runtime system written in C++ and for C++ applications. Specx allows parallelizing over distributed computing nodes and exploit CPUs and GPUs jointly. It is easy to use and provides advanced features such as scheduler customization and execution trace visualization.

We plan to improve Specx by providing a scheduler made for heterogeneous computing node, create new speculative execution model, add conditional tasks, and improve the compilation error handling.

7 Acknowledgement

We used the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Universite de Bordeaux, Bordeaux INP and Conseil Regional d'Aquitaine [5].

This work has been funded by the Inria ADT project SPETABARU-H, and the ANR National project AUTOSPEC (ANR-21-CE25-0009).

References

[1] Emmanuel Agullo, Olivier Aumage, Berenger Bramas, Olivier Coulaud, and Samuel Pitoiset. Bridging the gap between OpenMP and task-based runtime systems for the fast multipole method. *IEEE Transactions on Parallel and Distributed Systems, 28(10), 2794–2807*, 2017.

[2] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. Achieving High Performance on Supercomputers with a Sequential Task-based Programming

[5] https://www.plafrim.fr

Model. Research Report RR-8927, Inria Bordeaux Sud-Ouest ; Bordeaux INP ; CNRS ; Université de Bordeaux ; CEA, June 2016.

[3] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Paul Thibault. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2017.

[4] Emmanuel Agullo, Berenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-based FMM for heterogeneous architectures. *Concurrency and Computation: Practice and Experience*, 28(9):2608–2629, 2016.

[5] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. *ACM Trans. Math. Softw.*, 43(2):13:1–13:22, August 2016.

[6] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[7] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[8] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[9] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[10] Intel Threading Building Blocks. https://www.threadingbuildingblocks.org/.

[11] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[12] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[13] Bramas. Increasing the degree of parallelism using speculative execution in task-based runtime systems. *PeerJ Computer Science*, 5:e183, March 2019.

[14] Léopold Cambier, Yizhou Qian, and Eric Darve. Tasktorrent: a lightweight distributed task-based runtime system in c++. In *2020 IEEE/ACM 3rd Annual Parallel Applications Workshop: Alternatives To MPI+ X (PAW-ATM)*, pages 16–26. IEEE, 2020.

[15] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[16] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[17] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

[18] Kallia Chronaki, Marc Casas, Miquel Moreto, Jaume Bosch, and Rosa M. Badia. Taskgenx: A hardware-software proposal for accelerating task parallelism. In Rio Yokota, Michèle Weiland, David Keyes, and Carsten Trinitis, editors, *High Performance Computing*, pages 389–409, Cham, 2018. Springer International Publishing.

[19] Pierre-Nicolas Clauss and Jens Gustedt. Iterative computations with ordered read–write locks. *Journal of Parallel and Distributed Computing*, 70(5):496–504, 2010.

[20] Michel Cosnard and Michel Loi. Automatic task graph generation techniques. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, volume 2, pages 113–122 vol.2, Jan 1995.

[21] Anthony Danalis, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. Ptg: an abstraction for unhindered parallelism. In *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 21–30. IEEE, 2014.

[22] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters*, 21(02):173–193, 2011.

[23] Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. A survey on thread-level speculation techniques. *ACM Comput. Surv.*, 49(2), June 2016.

[24] Clément Flint, Ludovic Paillat, and Bérenger Bramas. Automated prioritizing heuristics for parallel task graph scheduling in heterogeneous computing. *PeerJ Computer Science*, 8:e969, September 2022.

[25] B. B. Fraguela and D. Andrade. Easy dataflow programming in clusters with upc++ depspawn. *IEEE Transactions on Parallel and Distributed Systems*, 30(6):1267–1282, 2019.

[26] Andrew Garmon, Vinay Ramakrishnaiah, and Danny Perez. Resource allocation for task-level speculative scientific applications: A proof of concept using parallel trajectory splicing. *Parallel Computing*, 112:102936, 2022.

[27] Thierry Gautier, Joao VF Lima, Nicolas Maillard, and Bruno Raffin. XKaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1299–1308. IEEE, 2013.

[28] Ruidong Gu and Michela Becchi. A comparative study of parallel programming frameworks for distributed gpu applications. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pages 268–273, 2019.

[29] Jérôme Gurhem and Serge G Petiton. A current task-based programming paradigms analysis. In *International Conference on Computational Science*, pages 203–216. Springer, 2020.

[30] Reazul Hoque and Pavel Shamis. Distributed task-based runtime systems - current state and micro-benchmark performance. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 934–941, 2018.

[31] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE Trans. Parallel Distrib. Syst.*, 33(6):1303–1320, jun 2022.

[32] Intel Cilk Plus. https://www.cilkplus.org/.

[33] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, pages 1–11, 2014.

[34] Laxmikant V Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *ACM Sigplan Notices*, volume 28, pages 91–108. ACM, 1993.

[35] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108, October 1993.

[36] Jeremy Kemp and Barbara Chapman. Mapping openmp to a distributed tasking runtime. In Bronis R. de Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta, editors, *Evolving OpenMP for Evolving Architectures*, pages 222–235, Cham, 2018. Springer International Publishing.

[37] S. K. Khatamifard, I. Akturk, and U. R. Karpuzcu. On approximate speculative lock elision. *IEEE Transactions on Multi-Scale Computing Systems*, 4(2):141–151, April 2018.

[38] Charles E Leiserson. The Cilk++ concurrency platform. In *46th ACM/IEEE Design Automation Conference, 2009. DAC'09*, pages 522–527. IEEE, 2009.

[39] Juan Manuel Martinez Caamaño, Manuel Selva, Philippe Clauss, Artyom Baloian, and Willy Wolff. Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones. *Concurrency and Computation: Practice and Experience*, 29(15):e4192, 2017. e4192 cpe.4192.

[40] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[41] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, July 2013.

[42] Josep M Perez, Rosa M Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151. IEEE, 2008.

[43] J. Salamanca, J. N. Amaral, and G. Araujo. Using hardware-transactional-memory support to implement thread-level speculation. *IEEE Transactions on Parallel and Distributed Systems*, 29(2):466–480, Feb 2018.

[44] Marc Snir, William Gropp, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *MPI–the Complete Reference: the MPI core*, volume 1. MIT press, 1998.

[45] Rupanshu Soi, Michael Bauer, Sean Treichler, Manolis Papadakis, Wonchan Lee, Patrick McCormick, Alex Aiken, and Elliott Slaughter. Index launches: scalable, flexible representation of parallel task groups. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–18, 2021.

[46] Anastasios Souris, Bérenger Bramas, and Philippe Clauss. Extending the Task Dataflow Model with Speculative Data Accesses. In *COMPAS 2023 - Conférence francophone d'informatique en Parallélisme, Architecture et Système*, Annecy (France), France, July 2023.

[47] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. *SIGARCH Comput. Archit. News*, 28(2):1–12, May 2000.

[48] Giuseppe Tagliavini, Daniele Cesarini, and Andrea Marongiu. Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight openmp tasking. *IEEE Transactions on Parallel and Distributed Systems*, 29(9):2150–2163, Sep. 2018.

[49] Samuel Thibault. *On Runtime Systems for Task-based Programming on Heterogeneous Platforms*. Habilitation à diriger des recherches, Université de Bordeaux, December 2018.

[50] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, et al. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, 74(4):1422–1434, 2018.

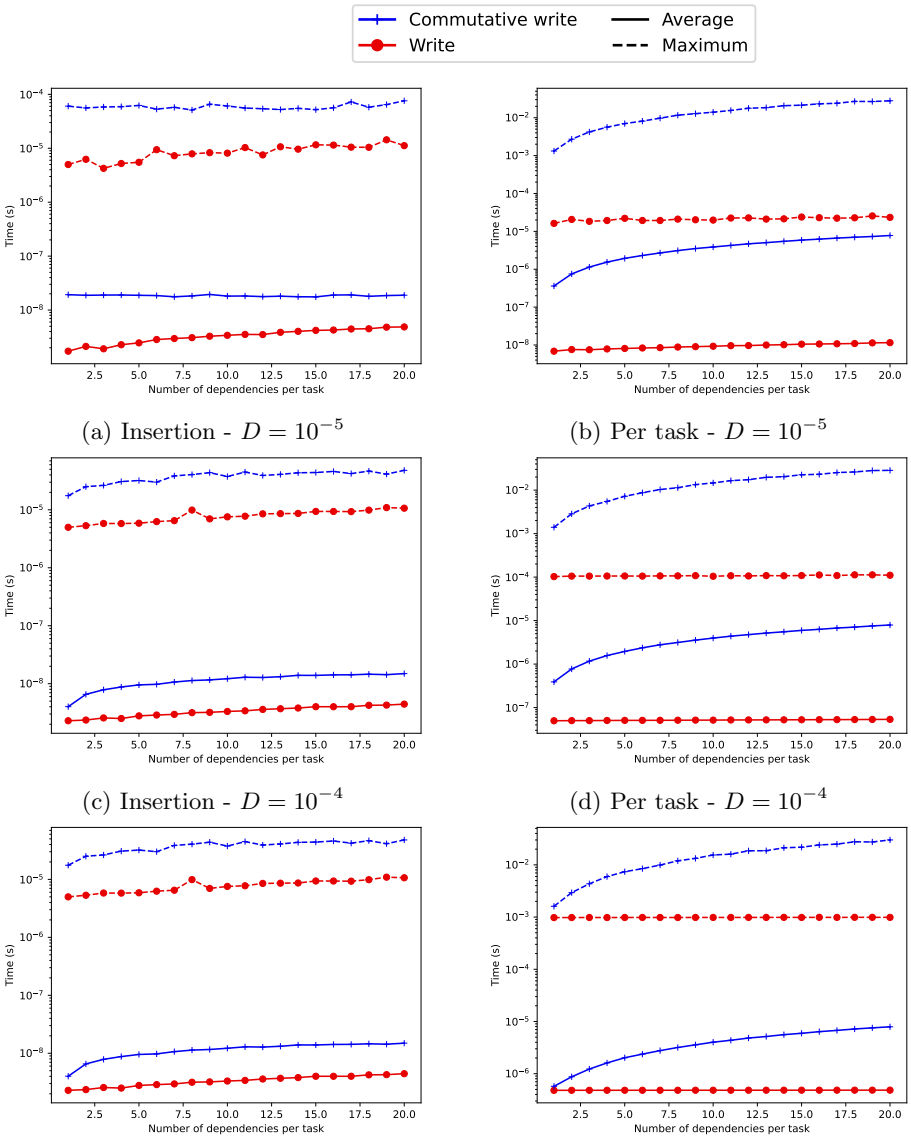
[51] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, pages 1–13, 2018.

[52] Martin Tillenius. Superglue: A shared memory framework using data versioning for dependency-aware task-based parallelization. *SIAM Journal on Scientific Computing*, 37(6):C617–C642, 2015.

[53] Christos Tsolakis, Polykarpos Thomadakis, and Nikos Chrisochoides. Tasking framework for adaptive speculative parallel mesh generation. *The Journal of Supercomputing*, 78(5):1–32, 2022.

[54] Afshin Zafari. Taskuniverse: A task-based unified interface for versatile parallel execution. In Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, and Konrad Karczewski, editors, *Parallel Processing and Applied Mathematics*, pages 169–184, Cham, 2018. Springer International Publishing.

[55] Afshin Zafari, Elisabeth Larsson, and Martin Tillenius. Ductteip: An efficient programming model for distributed task-based parallel computing. *Parallel Computing*, 90:102582, 2019.



(a) Insertion - $D = 10^{-5}$

(b) Per task - $D = 10^{-5}$

(c) Insertion - $D = 10^{-4}$

(d) Per task - $D = 10^{-4}$

(e) Insertion - $D = 10^{-3}$

(f) Per task - $D = 10^{-3}$

Figure 3: Estimation of the overheads for the *write* (•) and *commutative-write* (+) data accesses for different number of dependencies. We provide the maximum overhead reached (−−) and the average one (−). The overhead is given for picking a task O (right column) and the insertion I (left column).