PARALLEL PROGRAMMING...

Parallel Programming: Overview

SESSION 6/6

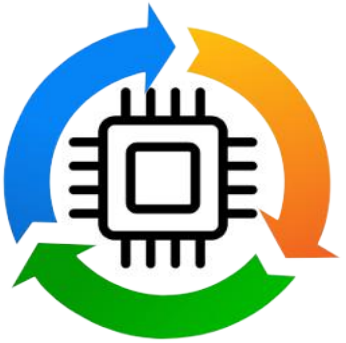
Programming Interface for parallel computing With SPECX

- What is SPECX ?
- Runtime Interface
- Data Dependency Interface
- Task Viewer Interface
- Future Developments

API Examples

What is SPECX?

양



SPECX

SPECX

- Shares many similarities with StarPU.
- Written in modern C++ (20).
- Task-based execution system.
- Able to also support speculative execution, which is the ability to execute tasks ahead of time if others are unsure about changing the data.

StarPU

- StarPU is a task scheduling library for hybrid architectures.
- Design systems in which applications are distributed across the machine, feeding all available resources into parallel tasks.
- Optimized heterogeneous scheduling, cluster communication, data transfers and replication between main memory and discrete memories

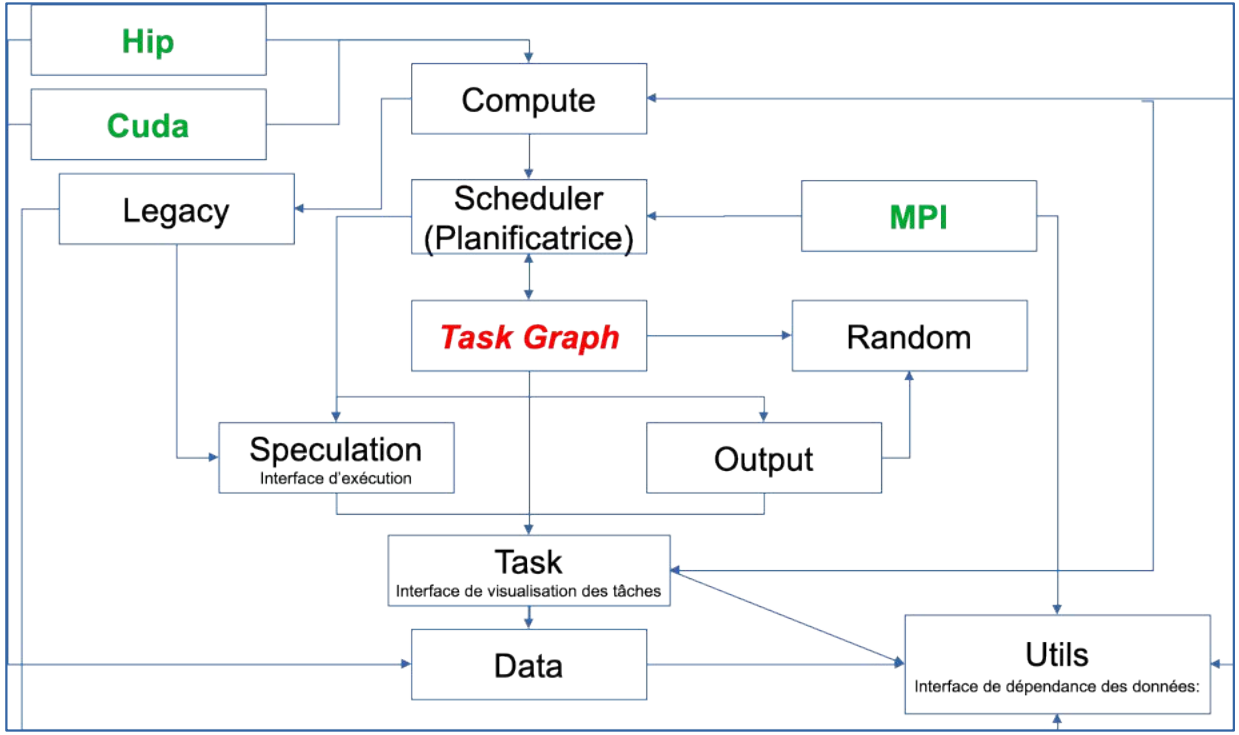
SPECX

Workflow

Execution interface: Provides functionality for creating tasks, task graphs and generating traces. Can be used to specify speculation model.

Data Dependency Interface: Forms a collection of objects that can be used to express data dependencies. Also provides wrapper objects that can be used to specify whether a given callable should be considered CPU or GPU code.

Task visualization interface: Specifies the ways to interact with the task object.



SPECX: Runtime Interface

The runtime's functionality is exposed through a class called **SpRuntime**.

This class provides task creation, task graph and trace generation facilities.

The **SpRuntime** class is templated on a non type parameter that can be used to specify which speculation model you want to use.

The runtime's contructor takes as a parameter the number of threads it should spawn.

By default the parameter is initialized to the number indicated by the **OMP_NUM_THREADS** environment variable.

SPECX: Runtime Interface

This method creates a new task and injects it into the runtime.

auto **task**([optional] **SpPriority** inPriority, [optional] **SpProbability** inProbability,
[optional] <DataDependencyTy> do..., <CallableTy> c) **(1)**

auto **task**([optional] **SpPriority** inPriority, [optional] **SpProbability** inProbability,
[optional] <DataDependencyTy> do..., **SpCpuCode**(<CallableTy> c1), [optional] **SpGpuCode**(<CallableTy> c2)) **(2)**

The inPriority parameter specifies a priority for the task.

The inProbability parameter is an object used to specify the probability with which the task may write to its maybe-written data dependencies.

In overload **(1)** the callable is passed as is to the task call. It will implicitly be interpreted by the runtime as CPU code.

In overload **(2)** the callable c1 is explictly tagged as CPU code by being wrapped inside a SpCpuCode object.

Overload **(2)** additionally permits the user to provide a GPU version of the code

When both CPU and GPU versions of the code are provided, the runtime will decide at runtime which one of the two to execute.

SPECX: Runtime Interface

```
Type1 v1; Type2 v2;

runtime.task(SpRead(v1), SpWrite(v2), [] (const Type1 &paramV1, Type2 &paramV2)
{
   if(paramV1.test()) {
      paramV2.set(1);
   }else {
      paramV2.set(2);
   }
});
```

Parameters corresponding to a SpRead data dependency object should be declared const.

Code inside the callable must be referring to parameter names rather than the original variable names.

In the example given above, code in the lambda body is referring to the names paramV1 and paramV2 to refer to v1 and v2 data values rather than v1 and v2.

You should not capture v1 and v2 by reference and work with v1 and v2 directly.

SPECX: Runtime Interface

void **setSpeculationTest**(std::function<bool(int,const SpProbability&)> inFormula)

This method sets a predicate function that will be called by the runtime whenever a speculative task is ready to be put in the queue of ready tasks .

The predicate returns a boolean. Reciprocally a return value of false means the speculative task and all of its dependent speculative tasks should be disabled.

If no speculation test is set in the runtime, the default behavior is that a speculative task and all its dependent speculative tasks will only be enabled if at the time the predicate is called no other tasks are ready to run.

SPECX: Runtime Interface

- void **waitAllTasks**()
  - *This method is a blocking call which waits until all the tasks that have been pushed to the runtime up to this point have finished.*
- void **waitRemain**(const long int windowSize)
  - *This method is a blocking call which waits until the number of still unprocessed tasks becomes less than or equal to windowSize.*
- void **stopAllThreads**()
  - *This method is a blocking call which causes the runtime threads to exit. The method expects all tasks to have already finished, therefore you should always call waitAllTasks() before calling this method.*
- int **getNbThreads**()
  - *This method returns the size of the runtime thread pool (in number of threads).*
- void **generateDot**(const std::string& outputFilename, bool printAccesses)
  - *This method will generate the task graph corresponding to the execution in dot format.*

SPECX: Data Dependency Interface

The data dependency interface forms a collection of objects that can be used to express data dependencies.

Data dependency objects : Specifying data dependencies boils down to constructing the relevant data dependency objects from the data lvalues.

Scalar data

- **SpRead(**x) *//Specifies a read dependency on x. Reads are ordered by the runtime with respect to writes, maybe-writes, commutative writes and atomic writes.*
- **SpWrite**(x) *//Specifies a write dependency on x indicating that the data x will be written to with 100% certainty. Multiple successive write requests to given data x will be fulfilled one after the other in the order they were emitted in at runtime. Writes are ordered by the runtime with respect to reads, writes, maybe-writes, commutative writes and atomic writes.*
- **SpMaybeWrite**(x) *//Specifies a maybe-write dependency indicating that the data x might be written to, i.e. it will not always be the case (writes might occur with a certain probability).*

SPECX: Data Dependency Interface

- **SpCommutativeWrite**(x) *//Specifies a commutative write dependency on x, i.e. writes that can be performed in any order.*
- *Multiple successive commutative write requests will be fulfilled one after the other in any order : while a commutative write request cw1 on data x is currently being serviced, all immediately following commutative write request on data x will be put on hold.*

- **SpAtomicWrite**(x) *//Specifies an atomic write dependency on x.*
- *Atomic write requests are always fulfilled by default, i.e. an atomic write request awr2 on data x immediately following another atomic write request awr1 on data x does not have to wait for awr1 to be fulfilled in order to be serviced.*
- *Multiple successive atomic writes will be performed in any order. The atomic writes will be committed to memory in whatever order they will be committed at runtime, the point is that the Specx runtime does not enforce an order on the atomic writes.*

SPECX: Data Dependency Interface

Non scalar data

We also provide analogous contructors for aggregates of data values from arrays :

- **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)

Wrapper objects for callables

We provide two wrapper objects for callables whose purpose is to tag a callable to inform the runtime system of whether it should interpret the given callable as CPU or GPU code:

- **SpCpuCode** (<CallableTy> c)
- **SpGpuCode** (<CallableTy> c)

SPECX: Task Viewer Interface

Main methods available on task objects returned by task calls

```
bool isOver()      //Returns true if the task has finished executing.

void wait()        //This method is a blocking call which waits until the task is finished.

<ReturnType> getValue()    // This method is a blocking call which retrieves the result value of the task.

void setTaskName(const std::string& inTaskName)   //Assigns the name in TaskName to the task.
                                                  // This change will be reflected in debug printouts, task graph
                                                  // and trace generation output.

std::string getTaskName()   //Retrieves the name of the task.
```

Nota: Speculative versions of tasks will have an apostrophe appended to their name.

SPECX: Exemples

SPECX: Future Developments

Future developments

- The main objective is to reduce the calculation times,
- To manage the use of the different calculation resources, the different typical workloads, in particular in the case of multicore machines equipped with several acceleration machines.
- Plan to separate thread management from execution. To change the prototype of the predicate, to be able to consider additional data or different to make the decision.
- Develop decision graphs to optimize available hybrid resources (CPU, GPU, GPGPU, TPU,...) to increase computational speed for given problems.
- To provide effective and high -performance tools to the user.

Thank you for your attention !