PARALLEL PROGRAMMING...

By Patrick Lemoine 2023.

Parallel Programming: Overview

SESSION 6/6

GOAL

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 and advanced C++ (20).
- Task-based execution system.

i.e. : 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

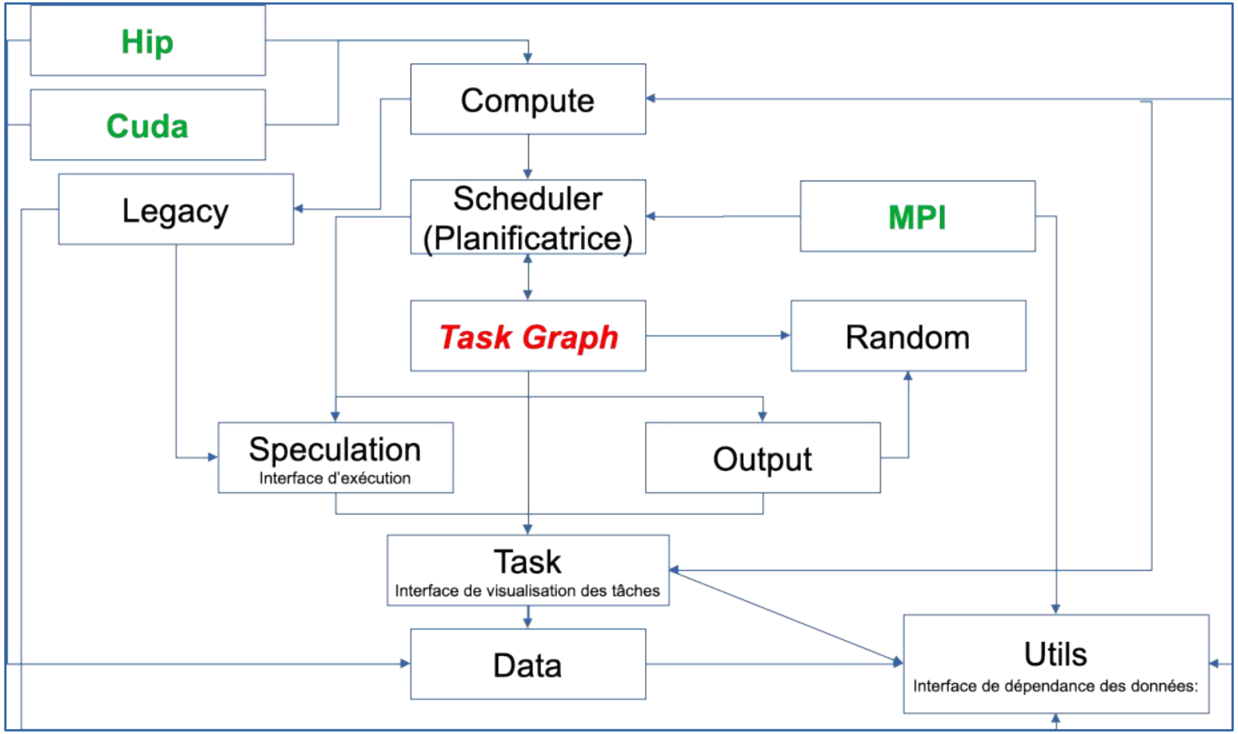
Runtime interface: Provides functionality for creating tasks, task graphs, and generating traces.

Used to specify a speculation model.

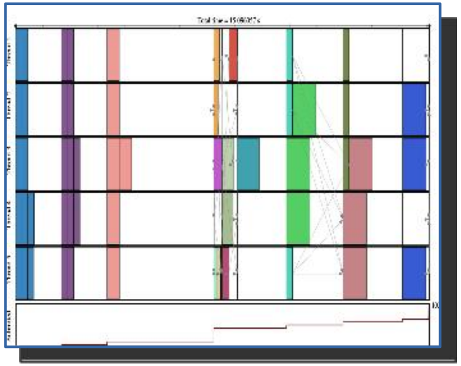
Data Dependency Interface: Forms a collection of objects that can be used to express data dependencies.

Provides "wrapper" that can be used to specify whether a given callable should be considered CPU or GPU code.

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

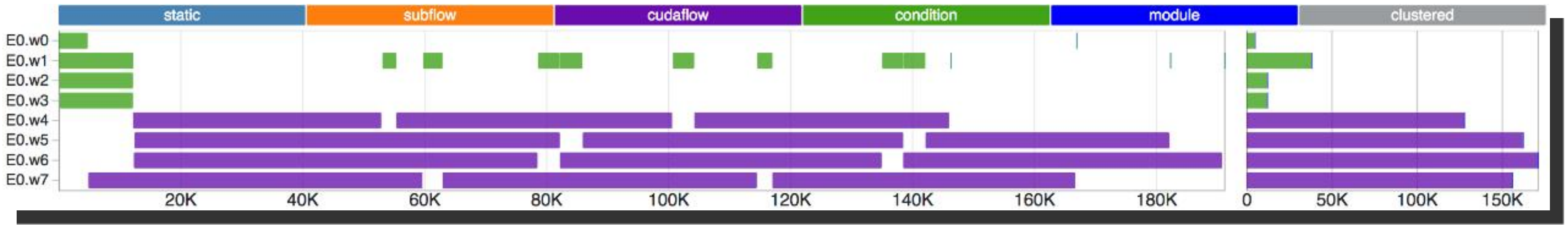


SPECX: Runtime Interface



The *runtime...*

- Provides features:
  - creating tasks.
  - task graph.
  - generation of traces.
- Can be used for**specify speculation model**.
- Its constructor takes as parameter the**number of threads**that it must generate.

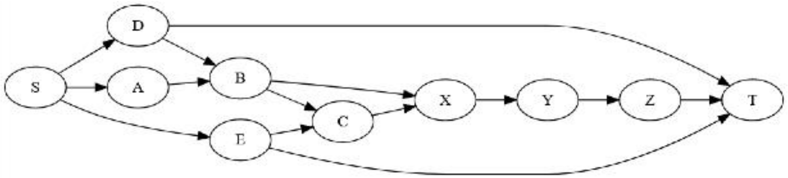


SPECX: Runtime Interface


```
Type1v1;Type2v2;

runtime.task(SpRead(v1),SpWrite(v2),
        [ ](const Type1¶mV1,Type2¶mV2)
        {
         If (paramV1.test()) { paramV2.set(1); } else { paramV2.set(2); }
        }
        );
```

Parameters corresponding to **SpRead**must be declared**const.**

- Code inside the callable should reference parameter names rather than the original variable names.



Def: A callable object is something that can be called like a function.

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)**

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

(2) the callable c1 is explicitly 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

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.

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 .

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**()
  - *Waits until all the tasks that have been pushed to the runtime up to this point have finished.*
- void **waitRemain**(const long int windowSize)
  - *Waits until the number of still unprocessed tasks becomes less than or equal to windowSize.*
- void **stopAllThreads**()
  - *The method expects all tasks to have already finished, therefore you should always*
  - ***call waitAllTasks() before** calling this method.*
- int **getNbThreads**()
  - *Returns the size of the runtime thread pool (in number of threads).*
- void **generateDot**(const std::string& outputFilename, bool printAccesses)
  - *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.

Scalar data

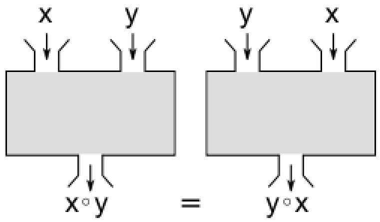
- **SpRead(**x) *//Reads are ordered by the runtime with respect to writes, maybe-writes, commutative writes and atomic writes.*

- **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 orderthey 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)
  - *Multiple successive commutative write requests will be fulfilled one after the other in any order*



- **SpAtomicWrite**(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)

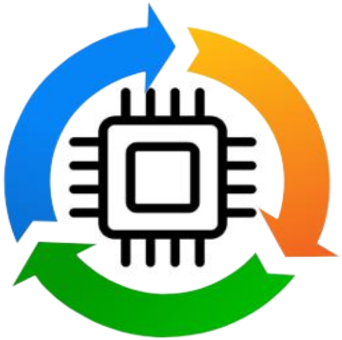
SPECX: Task Viewer Interface

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.
```

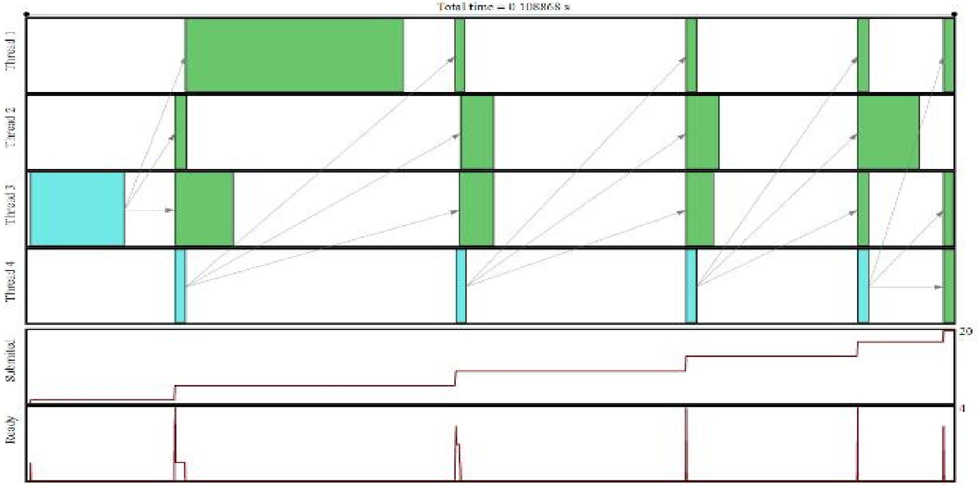
Nota: Speculative versions of tasks will have an apostrophe appended to their name.

Examples



SPECX: Exemple Heap Buffer

```
#include <iostream>

#include "Data/SpDataAccessMode.hpp"
#include "Utils/SpUtils.hpp"
#include "Task/SpTask.hpp"
#include "Legacy/SpRuntime.hpp"
#include "Utils/SpBufferDataView.hpp"
#include "Utils/SpHeapBuffer.hpp"
#include "Utils/small_vector.hpp"

int main(){
    const int NumThreads = SpUtils::DefaultNumThreads();
    SpRuntime runtime(NumThreads);

    SpHeapBuffer<small_vector<int>> heapBuffer;

    for(int idx = 0 ; idx < 5 ; ++idx){
        auto vectorBuffer = heapBuffer.getNewBuffer();

        runtime.task(SpWrite(vectorBuffer.getDataDep()),
          [](SpDataBuffer<small_vector<int>> /*vector*/){
        });

        for(int idxSub = 0 ; idxSub < 3 ; ++idxSub){
            runtime.task(SpRead(vectorBuffer.getDataDep()),
                         [](const SpDataBuffer<small_vector<int>> /*vector*/){
            });
        }
    }
```




```
    runtime.waitAllTasks();

    runtime.stopAllThreads();

    // We generate the task graph corresponding to the execution
    runtime.generateDot("Result.dot", true);

    // We generate an Svg trace of the execution
    runtime.generateTrace("Result.svg");

    return 0;
}
```

SPECX: Exemple Speculation Model

```
#include "Task/SpPriority.hpp"
#include "Task/SpProbability.hpp"
#include "Legacy/SpRuntime.hpp" ……….

[[maybe_unused]] const size_t seedSpeculationSuccess = 42;
[[maybe_unused]] const size_t seedSpeculationFailure = 0;
const size_t seed = seedSpeculationSuccess;

int main([[maybe_unused]] int argc, [[maybe_unused]] char *argv[]){
// First we instantiate a runtime object and we specify that the runtime should use
// speculation model 2.

/const int NumThreads = SpUtils::DefaultNumThreads();
//SpRuntime runtime(NumThreads);

   SpRuntime<SpSpeculativeModel::SP_MODEL_3> runtime;

// Next we set a predicate that will be called by the runtime each time a speculative
// task becomes ready to run. It is used to decide if the speculative task should be
// allowed to run.

   runtime.setSpeculationTest(
   []([[maybe_unused]] const int nbReadyTasks,
      [[maybe_unused]] const SpProbability& meanProbability) -> bool {
         return true; // Here we always return true, this basically means
                      // that we always allow speculative tasks to run
                      // regardless of runtime conditions.
      }
   );
int a = 41, b = 0, c = 0; int value;

// We create our first task. We are specifying that the task will be reading from a.
// The task will call the lambda given as a last argument to the call.
// The return value of the task is the return value of the lambda.

auto task1 = runtime.task(SpRead(a),
     [](const int& inA) -> int { return inA + 1;}
);

// Here we set a custom name for the task.
task1.setTaskName("First-task");

// Here we wait until task1 is finished and we retrieve its return value.
b = task1.getValue();
```

SPECX: Exemple Speculation Model

```
// Next we create a potential task, i.e. a task which might write to
// some data.
// In this case the task may write to "a" with a probability of 0.5.
// Subsequent tasks will be allowed to speculate over this task.
// The task returns a boolean to inform the runtime of whether or
// not it has written to its maybe-write data dependency a.

std::mt19937_64 mtEngine(seed); //Pseudo Random generator 32 bit
                                //numbers with a state size of 19937

std::uniform_real_distribution<double> dis01(0,1); //Produces random
//floating-point values, uniformly distributes.

auto task2 = runtime.task(
    SpPriority(0), SpProbability(0.5), SpRead(b),SpPotentialWrite(a),

    [dis01, mtEngine] (const int &inB, int &inA) mutable -> bool

    {

        double val = dis01(mtEngine);

        If( inB == 42  && val < 0.5) { inA = 43; return true; }

        return false;

    }
);
```

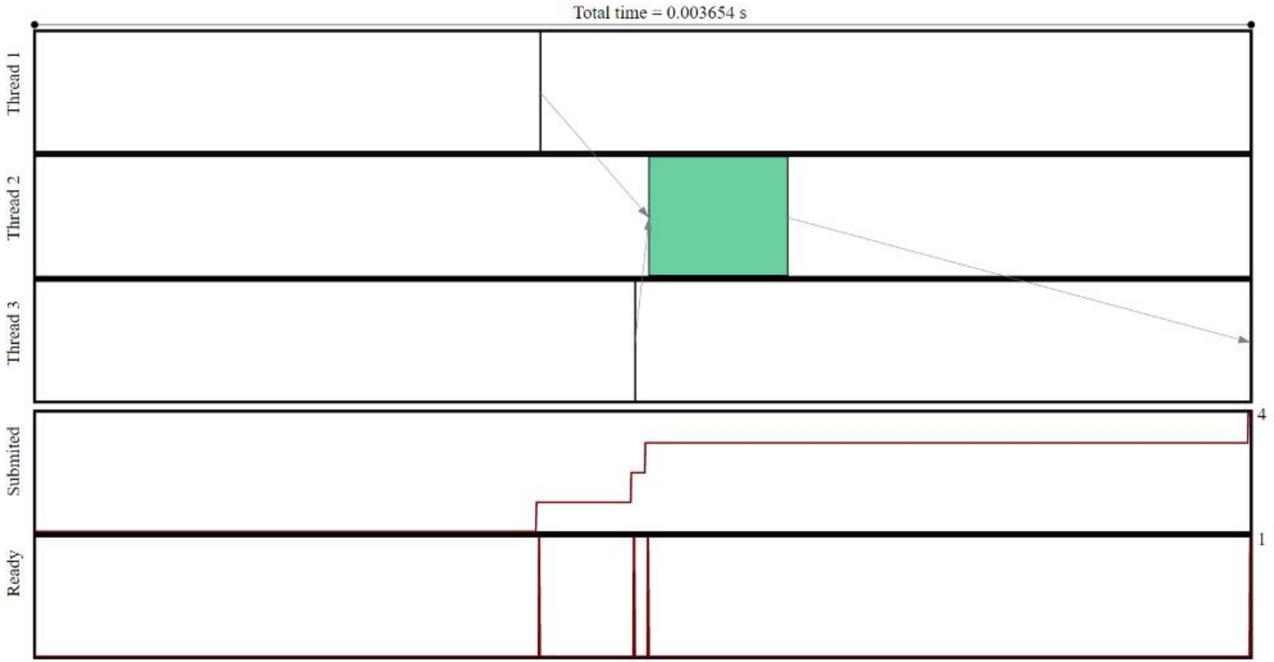
```
task2.setTaskName("Second-task");
value=task1.getValue(); printf("value task1=%i\n",value);
value=task2.getValue();  printf("value task2=%i\n",value);

 auto task3 =runtime.task( SpRead(a), SpWrite(c),
    [ ] (const int &inA, int &inC)
    {
        if(inA == 41) { inC = 1;} else { inC = 2;}
    }
);

task3.setTaskName("Final-task");

// We wait for all tasks to finish
runtime.waitAllTasks();

// We make all runtime threads exit
runtime.stopAllThreads();

assert((a == 41 || a == 43) && b == 42 && (c == 1 || c == 2) && "Try again!");
// We generate the task graph corresponding to the execution
runtime.generateDot("Result.dot", true);

// We generate an Svg trace of the execution
runtime.generateTrace("Result.svg");

return 0;
}
```

SPECX: Exemples

Result.dot

```
digraph G {
                0 -> 2
                0 [label="First-task
READ 0x7ffe811724f0
"];
                1 -> 2
                1 [label="sp-copy
WRITE 0x55f486e5efe0
READ 0x7ffe811724f0
"];
                3 [label="Final-task
READ 0x7ffe811724f0
WRITE 0x7ffe811724f8
"];
                2 -> 3
                2 [label="Second-task
READ 0x7ffe811724f4
POTENTIAL_WRITE 0x7ffe811724f0
"];
}
```

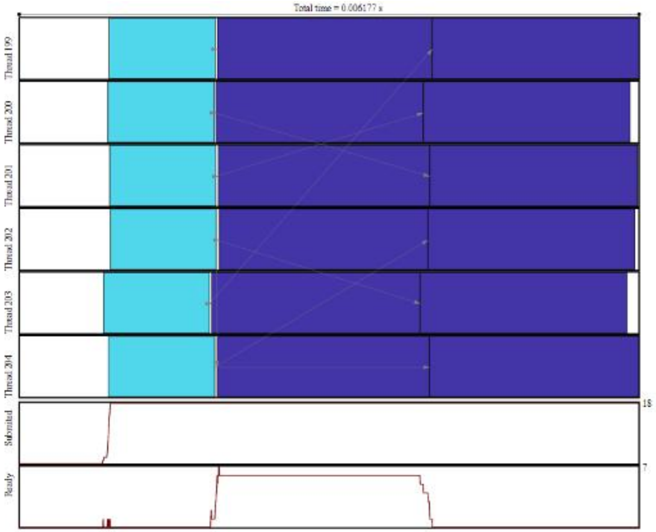
Result.svg



SPECX: Exemple VectorBuffer

```
{
    const int initVal = 1;
    int writeVal = 0;

    NumThreads=6;
    SpRuntime My_Runtime2(NumThreads);

    small_vector<int> vs;
    std::cout << "std::allocator<int>:"<< '\n'
              << "  sizeof (vs):     " << sizeof (vs)          << '\n'
              << "  Maximum size:    " << vs.max_size ()       << "\n\n";

    SpHeapBuffer<small_vector<int>> heapBuffer;

    int valueN=0; int valueM=0;

    for(int idx = 0 ; idx < 6 ; ++idx){
        auto vectorBuffer = heapBuffer.getNewBuffer();

        My_Runtime2.task(SpWrite( vectorBuffer.getDataDep()) ,

                [&](SpDataBuffer<small_vector<int>> ) mutable
                {
                    valueN=idx;
                    usleep(1000);
                }

        ).setTaskName("Write Vector Buffer");

        for(int idxSub = 0 ; idxSub < 2 ; ++idxSub){
            My_Runtime2.task(SpRead( vectorBuffer.getDataDep() ),

                    [=] (const SpDataBuffer<small_vector<int>>)
                    {
                     ….
                     usleep(2000);
                    }
            ).setTaskName("Read Vector Buffer");
        } //End For idxSub
    } //End For idx

    My_Runtime2.waitAllTasks();
    My_Runtime2.stopAllThreads();
    My_Runtime2.generateDot("Result.dot",true);
    My_Runtime2.generateTrace("Result.svg");
}
```

Result.svg



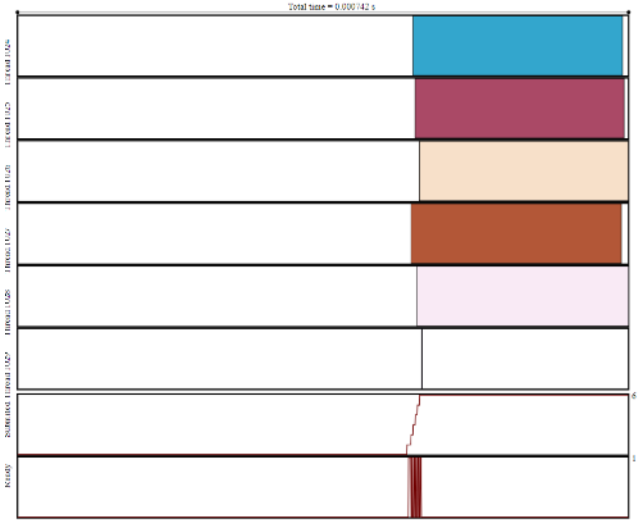
SPECX: Examplevscalculate the value ofIP

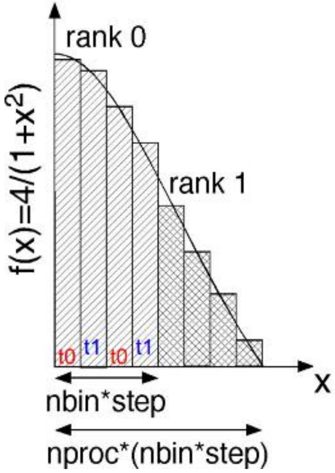
```
int nbThreads = std::min(6,SpUtils::DefaultNumThreads());
SpRuntimeruntime(nbThreads);

long int nbN=1000000;
int sizeBlock=nbN/nbThreads;
int diffBlock=nbN-sizeBlock*nbThreads;
double h=1.0/double(nbN);
double integralValue=0.0;
std::vector<double> valuesVec(nbThreads,0.0);

for(int k1 = 0; k1 < nbThreads; ++k1){
int vkBegin=k1*sizeBlock;
int vkEnd=(k1+1)*sizeBlock;
if ((k1==nbThreads-1) && (k1>0) && (diffBlock>0)) { vkEnd=vkBegin+diffBlock; }
int threadid = k1;
runtime.task(
SpWrite(valuesVec.at(threadid)),
[h,vkBegin,vkEnd](double& s) -> bool {
double sum=0.0; doublex;
for(int j=vkBegin;j<vkEnd;j++)
{
x=h*double(j);
sum+=4.0/(1.0+x*x);
}
s=sum;
return true;
}
).setTaskName("Op("+std::to_string(k1)+")");
}
```

```
…….
//Sum of vector elements
integralValue=h*std::reduce(valuesVec.begin(),valuesVec.end());
double DeltaError=std::abs(M_PI-integralValue);
```

OBJECTIVE: The following code calculates the number π using a numerical evaluation of an integral by a rectangle method.

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \cong \Delta \sum_{i=0}^{N-1} \frac{4}{1+x_i^2}$$





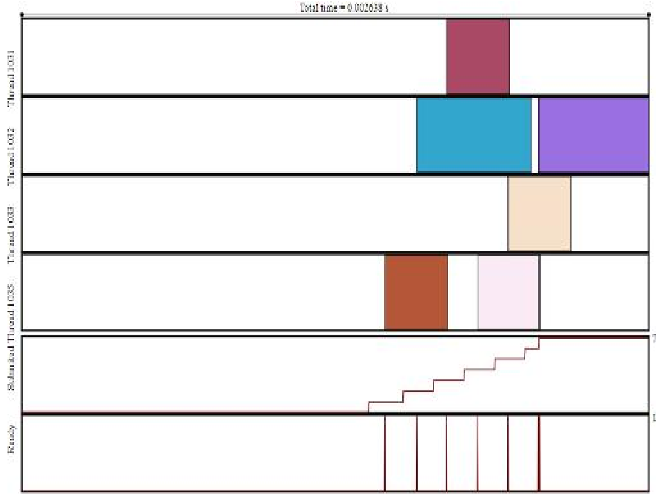
SPECX: Examplevscalculate the value ofIP

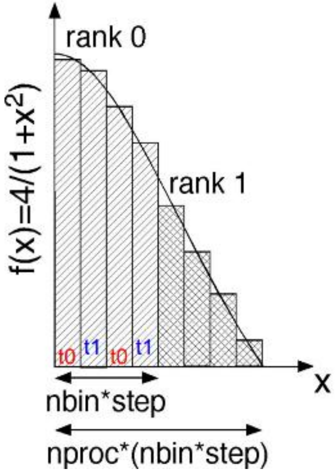
```
int nbThreads = std::min(6,SpUtils::DefaultNumThreads());
 SpRuntimeruntime(nbThreads);
long int nbN=1000000;
int sizeBlock=nbN/nbThreads;
int diffBlock=nbN-sizeBlock*nbThreads;
double h=1.0/double(nbN);
double integralValue=0.0;
//We build a list of numbers.
std::vector<int> v(boost::counting_iterator<int>(0), boost::counting_iterator<int>(nbN));
…...

  //We calculate the integral
   std::vector<double> valuesVec(size,0.0);
   for(int k1 = 0; k1 < size; ++k1){
   int threadid = k1;
   auto const & ra = ranges[k1];
    runtime.task(
   SpWrite(valuesVec.at(threadid)),
   [h,ra](double& s) -> bool {
   double sum=0.0; doublex;
   for (int i=0;i<ra.size();i++)
   {
   x=h*double(ra.at(i));
   sum+=4.0/(1.0+x*x);
   }
   s=sum;
   return true;
   }
   ).setTaskName("Op("+std::to_string(k1)+")");
   }
```

```
…….
//Sum of vector elements
integralValue=h*std::reduce(valuesVec.begin(),valuesVec.end());
double DeltaError=std::abs(M_PI-integralValue);
```

OBJECTIVE: The following code calculates the number π using a numerical evaluation of an integral by a rectangle method.

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \cong \Delta \sum_{i=0}^{N-1} \frac{4}{1+x_i^2}$$





SPECX: Exemple with promise

```
{
   NumThreads=2;
   SpRuntime My_Runtime3(NumThreads);

   //SpRuntime<SpSpeculativeModel::SP_MODEL_2> My_Runtime3;
   //cout<<":"<<My_Runtime3.getValue()<<"\n";

   My_Runtime3.setSpeculationTest(
      [] (const int /*inNbReadyTasks*/,
         const SpProbability& /*inProbability*/) -> bool
      {
         // Always speculate
         return true;
      }
   );

   int val = 0;
   std::promise<int> promise3; //the promise that append thread must fulfill.

    My_Runtime3.task( SpRead(val),

      [&promise3] (const int& /*valParam*/)
      {
         usleep(100);
         promise3.get_future().get(); //Returns a future object that has the same associated
                                      //asynchronous state as this promise object.
      }
   ).setTaskName("First task");

   for(int idx = 0; idx < 1; idx++) {
     My_Runtime3.task( SpWrite(val),
        [ ] (int& valParam)
        {
         cout<<"CTRL Val in certain task="<<valParam<<"\n";
         usleep(500);
        }
     ).setTaskName("Certain task -- " + std::to_string(idx));
   }

   const int nbUncertainTasks = 6;

   for(int idx = 0 ; idx < nbUncertainTasks ; ++idx){
      My_Runtime3.task( SpPotentialWrite(val),
        [ ] (int& valParam) -> bool
         {
           usleep(1000);
           return true;
         }
      ).setTaskName("Uncertain task -- " + std::to_string(idx));
   }
```

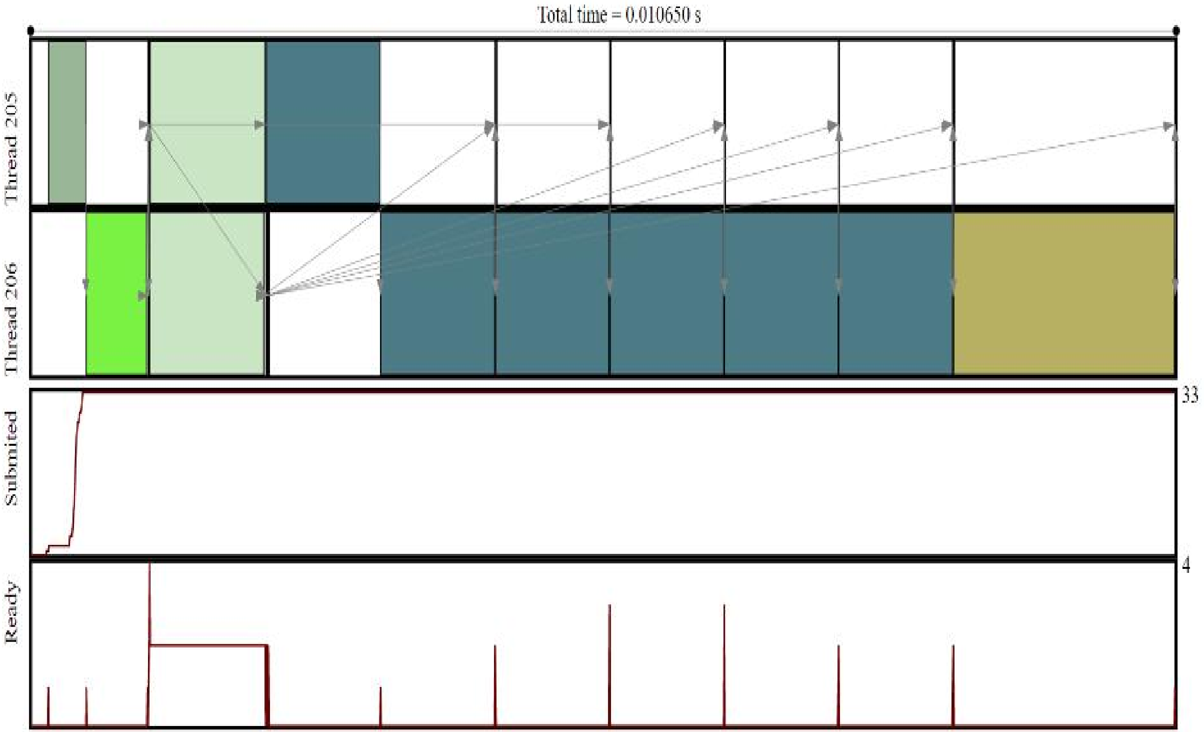
```
My_Runtime3.task(SpWrite(val),
            [ ] ([[maybe_unused]] int& valParam)
            {
              usleep(2000);
            }
        ).setTaskName("Last-task");}
```

promise3.set_value(0); // This operation acquired a single mutex associated with the promise object when updating the promise object.

An exception is thrown if there is no shared state or if the shared state already stores a value or an exception. Calls to this function don't introduce data races with calls to get_future (so they don't need to synchronize with each other).

```
My_Runtime3.waitAllTasks();
My_Runtime3.generateDot("Result.dot",true);
My_Runtime3.generateTrace("Result.svg");

}
```

Result.svg



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 !