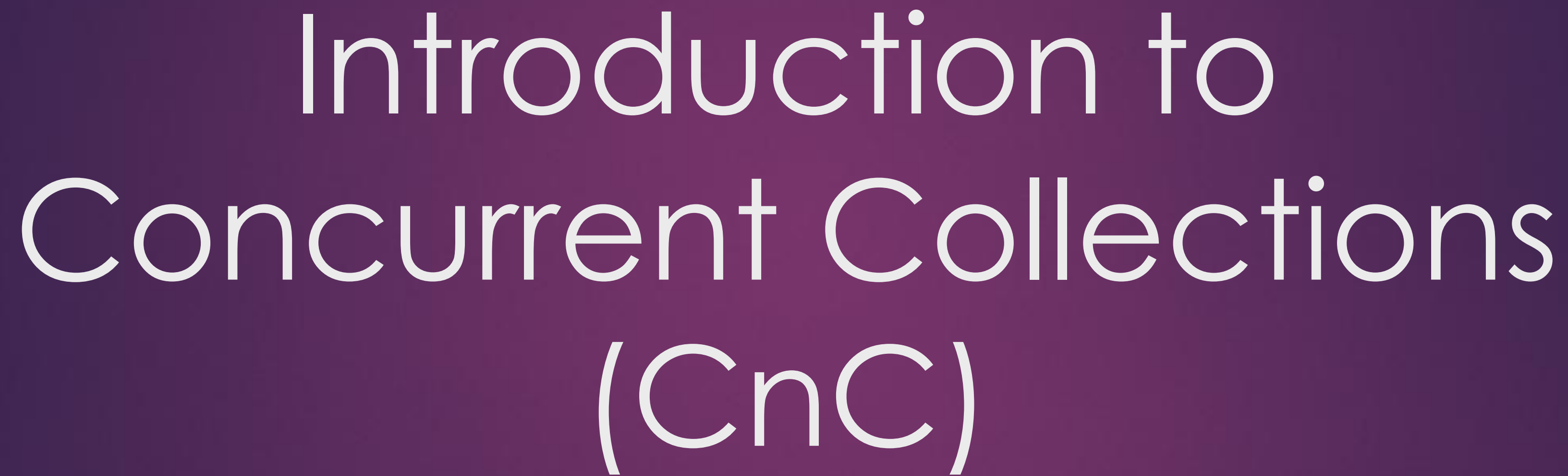

Introduction to
Concurrent Collections
(CnC)

Introduction - Motivation

- Why should chemists/physicists care about parallelism? Can they take benefit of parallelism without having knowledge of parallelism?
- The main motivation: **Targeting large community of non-professional programmers to use parallelism**. How?
  - By **Separation of concerns** between **application logic** *(domain expert* takes care of it*)* and **parallel implementation** *(tuning expert* takes care of it*)*. How?
    - By Avoiding explicit parallelism/threading.
    - Avoiding exposing too many of the hardware details.
    - Successful stories: **MapReduce**, **Nvidia CUDA, etc.**
  - CnC is a mean to reach such a purpose.

What is CnC?

- Stands for **Concurrent Collections**.
- Falls into the family of dataflow and stream-processing paradigms:
  - A program is a **graph of computation nodes**, **communicating** with one another.
  - Computations are called **step collections** and are **related by control** and **data dependencies**.
  - CnC is simultaneously a **dataflow-like parallel model** and a simple specification language that **facilitates communication** between **the domain** and **expert experts**.
    - Domain experts:
      - Takes care of application logic. E.g., Chemist/Physicist/etc
      - Having (almost) no knowledge about parallelism.
    - Tuning experts:
      - Given the maximum possible freedom to map the computation onto the target architecture.
      - Having (almost) no knowledge about the domain (chemistry/physics/etc)

Burgers with Fries and Pies for Dessert - CnC in a Nutshell

- (As a catering service,) Let's make Burgers and French Fries for serving the IACS opening ceremony …
- Ready-to-process ingredients: cut potatoes and prepared meat → get delivered to a service hatch.
  - In CnC, such hatches are called ***item-collections***, which store input/output data/items. So, out hungry guests can pick up the food from the output hatches.
- Tasks? Frying the potatoes and barbecuing the meat.
  - In CnC, such tasks are called ***step-collections***. The steps are basically just normal functions.
    - to be implemented mainly by the domain expert
    - Each instance of such step-collections is a function call with different parameters passed to it.

Burgers with Fries and Pies for Dessert - CnC in a Nutshell – (Cont)'

- Tasks? Frying the potatoes and barbecuing the meat.
  - Is there any dependency between these two steps? No. So, there is no ordering required.
- [**producer/consumer Dependency**] Let's assume university president asks to add mini cherry pies to the meal for dessert. Tasks for making mini-pie are: preparing the pies and baking them.
  - Is there any dependency between these two steps (prepare_pie and bake_pie)? Yes. So, there is producer/consumer dependency: the producer needs to be executed before the consumer can use the produced item.

Burgers with Fries and Pies for Dessert - CnC in a Nutshell – (Cont)''

- [**Controller/Controlee Dependency**] Let's assume guests are individuals and not all of them want the same menu. Some might want the full menu (a burger, fries and a pie). Some might prefer two burgers, no fries but a pie and so on.
  - So, let's first take orders (take_orders) first. When taking orders, we must assign a unique identifier (a tag) to each order so that we later know burgers, fries and pies are for which guest.
  - To communicate the orders with the kitchen, we use special bowls, one for each step ("barbecue_burger", "fry_potatoes", "bake_pie" and "prepare_pie").
  - Whenever a step needs to be executed for a given order, the waiter (controller) simply puts the corresponding guest-tag into the step's bowl.
  - Is there any dependency between "taking_order" step and other steps? Yes. In CnC, this is controller/controlee relationship. "taking_order" step decides which steps need to be executed.

Burgers with Fries and Pies for Dessert - CnC in a Nutshell – (Cont)”’

- Now, we have full specification:
  - Step collections: (barbecue_burger), (fry_potatoes), (bake_pie), (prepare_pie and (take_orders).
  - Data/item collections: [meat], [potatoes], [fries], [burgers], [pies]
  - Control tags: guest-id
  - Consumer/producer dependency: (prepare_pie) → [pies] → (bake_pie)
  - Controller/controlee dependency: (take_orders) ➔ xxx_bowls ➔ other steps
- The dependencies imply a partial ordering. This partial ordering allows the CnC runtime engine to determine a legal scheduling of the step instances (computation units), be it serial or parallel.
- Constraints:
  - Computation units must execute statelessly, e.g., they must not access or even alter any global data.
  - Data is immutable. Once put, data items can’t be altered. Instead of changing a value, in CnC, you put a new value with a new tag !

CnC Constructs

- CnC has three main collections. These collections and their relationships are defined statically. However, for each static collection, a set of dynamic instances is generated at runtime. Collections are :
  - **Step collections:** a step collection corresponds to a specific computation (a procedure) and its instances correspond to invocations of that procedure with different arguments/inputs.
  - **Data/item collections:** step collection dynamically reads/writes data collection instances.
  - **Control collections**: a control collection is said to *prescribe* a step collection – adding an instance to the control collection will cause a corresponding step instance to eventually execute with that control instance as input (REMEMBER: In CnC, step collections are NEVER called explicitly)

```
// control relationship: myCtrl prescribes instances of step
<myCtrl> :: (myStep);
// consume from myData, produce to myCtrl, myData
[myData] → (myStep) → <myCtrl>, [myData];
```

Fibonacci in CnC (1)

- Fibonacci computation is defined as follows:
  - Fib(n) = Fin(n-1) + Fib(n-2)
  - Defining a new data type as the values of Fibonacci grow very large:

```
typedef unsigned long long fib_type;
```

- **Identifying Computation Unit** (**step collection**):
  - There is only one computation unit which simply adds the values of the two previous Fibonacci numbers. Let's call it "fib_step" (to be defined later). Declaring an instance of such step is done as follows:

```
CnC::step_collection< fib_step > m_steps;
```

- **Determining the data entities** (**data/item collection**):
  - There is only one data item we seem to care about regarding Fibonacci computation: the Fibonacci number that we compute.

Fibonacci in CnC (2)

- **Determining the data entities** (**data/item collection**):
  - There is only one data item we seem to care about regarding Fibonacci computation: the Fibonacci number that we compute.
  - The only way in CnC to read values from a computation is through item-collections. Here, in case of recursive computation, as Fibonacci, we can use one item-collection for the intermediate results and for the final result. Defining an item-collection is straight forward:

```
CnC::item_collection< int, fib_type > m_fibs;
```

In this definition:

- The first-type argument is the tag-type for identifying each data-instance (just like traditional key/value pair, the value of our item is accessible (only) through its identifier, the tag).
- The second-type argument is the type of the data to be stored.

Fibonacci in CnC (3)

- **Defining the step collection fib_step:**
  - In CnC, the step collections are as a class/struct with an execute method which accepts two argument: a control tag and a second argument, which usually is a the context (to be defined later):

```
struct fib_step {
     // declaration of execute method goes here
     int execute( const int & tag, fib_context & c ) const;
};
```

  - Note that the execute(…) function should be "const" and is not allowed to have side-effects.
  - The control tag distinguishes between different execution instances of the same step.
  - In CnC literature, context is referred to as "graph". It brings together the different collections (to be defined later for this case study).

Fibonacci in CnC (4)

- **Determining the Control Tags** (**Control Collections**):
  - In CnC, steps are NEVER called explicitly. If a step needs to be executed with a given tag, this tag is put into a so called tag-collection. The tag-collection makes sure that the step gets executed eventually. So, for the Fibonacci case study, the definition is as follows:

```
CnC::tag_collection< int > m_tags;
```

- **The context:**
  - As mentioned, it is referred to as "graph". It brings together the different collections (tags, items and steps) by defining them as members.
  - Each content must be derived from a base class, which again is a template.

Fibonacci in CnC (5)

- **The context:**
  - As mentioned, it is referred to as "**graph**". It brings together the different collections (tags, items and steps) by defining them as members.
  - Each content must be derived from a base class, which again is a template.
  - For the Fibonacci case study, it is as follows:

```
// derive from CnC::context
struct fib_context : public CnC::context< fib_context >
{
   // the step collection for the instances of the compute-kernel
   CnC::step_collection< fib_step > m_steps;
   // item collection holding the fib number(s)
   CnC::item_collection< int, fib_type > m_fibs;
   // tag collection to control steps CnC::tag_collection< int > m_tags;
   // constructor
   fib_context();
};
```

Fibonacci in CnC (6)

- **The context constructor function:**
  - In the constructor of the context, we define the relations (producer/consumer and controller/controlee) between the different collections:
    - Producer/Consumer dependency is defined by calling consumes() and produces() functions.
    - For each tag which is put into a tag collection m_tags, a step from m_steps is executed. This is defined by invoking the method prescribe on m_tags.

```
// derive from CnC::context
struct fib_context : public CnC::context< fib_context >
{
   // the step collection for the instances of the compute-kernel
   CnC::step_collection< fib_step > m_steps;
   // item collection holding the fib number(s)
   CnC::item_collection< int, fib_type > m_fibs;
   // tag collection to control steps CnC::tag_collection< int > m_tags;
   // constructor
   fib_context();
};
```

```
fib_context::fib_context()
        : CnC::context< fib_context >(),
        // pass context to collection constructors
        m_steps( *this ),
        m_fibs( *this ),
        m_tags( *this )
{ // prescribe compute steps with this (context) as argument
    m_tags.prescribes( m_steps, *this );
    // step consumes m_fibs
    m_steps.consumes( m_fibs );
    // step also produces m_fibs
    m_steps.produces( m_fibs );
}
```

Fibonacci in CnC (7)

- **Writing the Step by the *domain expert*:**
  - For the step collection fib_step which has been defined as follows

```
struct fib_step {
   // declaration of execute method goes here
   int execute( const int & tag, fib_context & c ) const;
};
```

```
int fib_step::execute( const int & tag, fib_context & ctxt ) const {
   switch( tag ) {
      case 0 : ctxt.m_fibs.put( tag, 0 ); break;
      case 1 : ctxt.m_fibs.put( tag, 1 ); break;
      default :
         // get previous 2 results
         fib_type f_1; ctxt.m_fibs.get( tag - 1, f_1 );
         fib_type f_2; ctxt.m_fibs.get( tag - 2, f_2 );
         // put our result
         ctxt.m_fibs.put( tag, f_1 + f_2 );
   }
   return CnC::CNC_Success;
}
```

Fibonacci in CnC (8)

- **The Main Program:**

```
int main( int argc, char* argv[] ) {
   int n = 42;
   // eval command line args
   if( argc < 2 ) {
      std::cerr << "usage: " << argv[0] << " n\nUsing default value " << n << std::endl;
   } else n = atol( argv[1] );
   // create context
   fib_context ctxt;
   // put tags to initiate evaluation
   for( int i = 0; i <= n; ++i ) ctxt.m_tags.put( i );
   // wait for completion
   ctxt.wait();
   // get result
   fib_type res2;
   ctxt.m_fibs.get( n, res2 );
   // print result
   std::cout << "fib (" << n << "): " << res2 << std::endl;

   return 0;
}
```

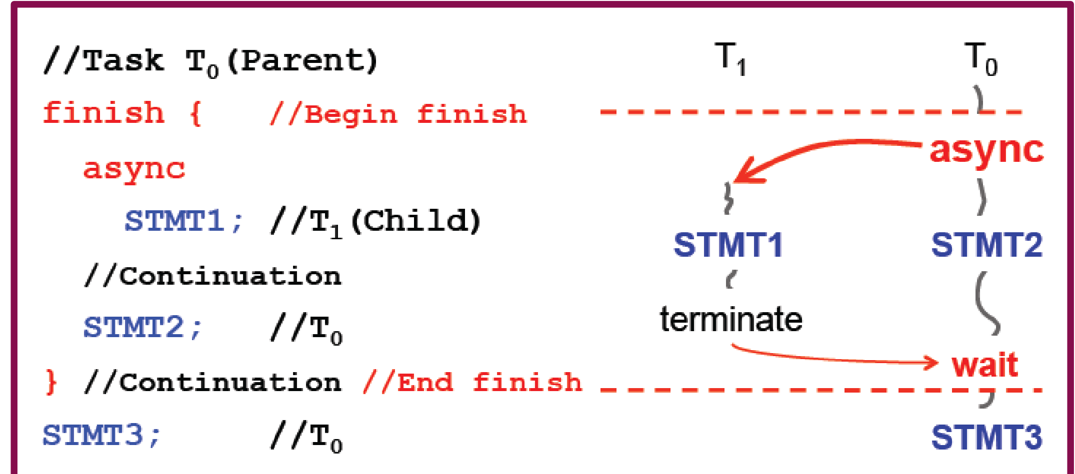
They explain that: You might have noticed that the fib(n-1) will not become available until a corresponding step-instances has been executed. Hence, it will not be sufficient to prescribe only the desired step-instances, that's why we need to put all tags up to that number.

Implementation

- Implementations of CnC need to provide a translator and runtime.
- They have implemented CnC for C++, Java, .NET and Haskell.
- For C++, they use C++ Threading Building Blocks (TBB) and also recently, Habanero-C.
- For Java, they uses Habanero-Java (an extension of X10 language).
- For Haskell, they use the work stealing features of the Glasgow Haskell Compiler to implement CnC.

Main Feature of Haabnero-C

- Habanero-C:
  - C-based task-parallel programming language developed at Rice University.
  - Main (parallelism) Feature:
    - The *async and finish* constructs, which define lightweight dynamic task creation and termination (originally defined in the X10 language):
      - The statement “async <stmt>” causes the parent task to crate a new child task to execute <stmt> asynchronously (i.e., before, after or in parallel) with the remainder of the parent task. So, in the figure, STMT1 in task T_1 can potentially execute in parallel with STMT2 in task T_0.
      - The statement “finish <stmt>” causes the parent task to execute <stmt> and then wait until all async tasks created within <stmt> have completed, including transitively spawned tasks. So, in the figure, child task T1 has completed executing STMT1 before T0 executes STMT3.



Further investigations regarding CnC

- Formal description of an execution semantics for CnC with a proof of determinism [3] :
  - Since parallelism provides inherent un-determinism in the execution, they have proved that the ultimate state of the two executions of the same program will lead to the same result (though the order of execution of the step collection instances might be different).
- Reasons for using Habanero-C and C++ TBB for the implementation.
- Other extensions of CnC such as HC-CnC [1]

References

- [1] Mapping a Data-Flow Programming Model onto Heterogeneous Platforms, Alina Sbirlea et al.
- [2] https://icnc.github.io/api/fib.html
- [3] Concurrent Collections, Zoran Budimlic, et al.