Intel® Concurrent Collections In Action

Cern, 7.May.2010
Frank Schlimbach
Mario Deilmann
Kath Knobe

Action Overview

- Overview
  - CnC textual syntax
  - API
- Install, CnC kit, environment etc.
- Hands on Fibonacci in CnC
  - Simple
  - Tuning
  - Distributed CnC
  - Ranges

Main features

- Domain expert oriented
  - Focus on application semantics
  - Parallelization expertise not required
- High scalability
  - No over-constraints of computations (no arbitrary serialization)
  - Only constraints are fine level data dependencies and control dependencies
- Deterministic and race-free
  - No overwriting of data
  - Computation units are functional (no side effects)
- General enough for all styles of parallelism
  - Task, data, pipeline, etc
- Portable
  - Interchangeable runtimes

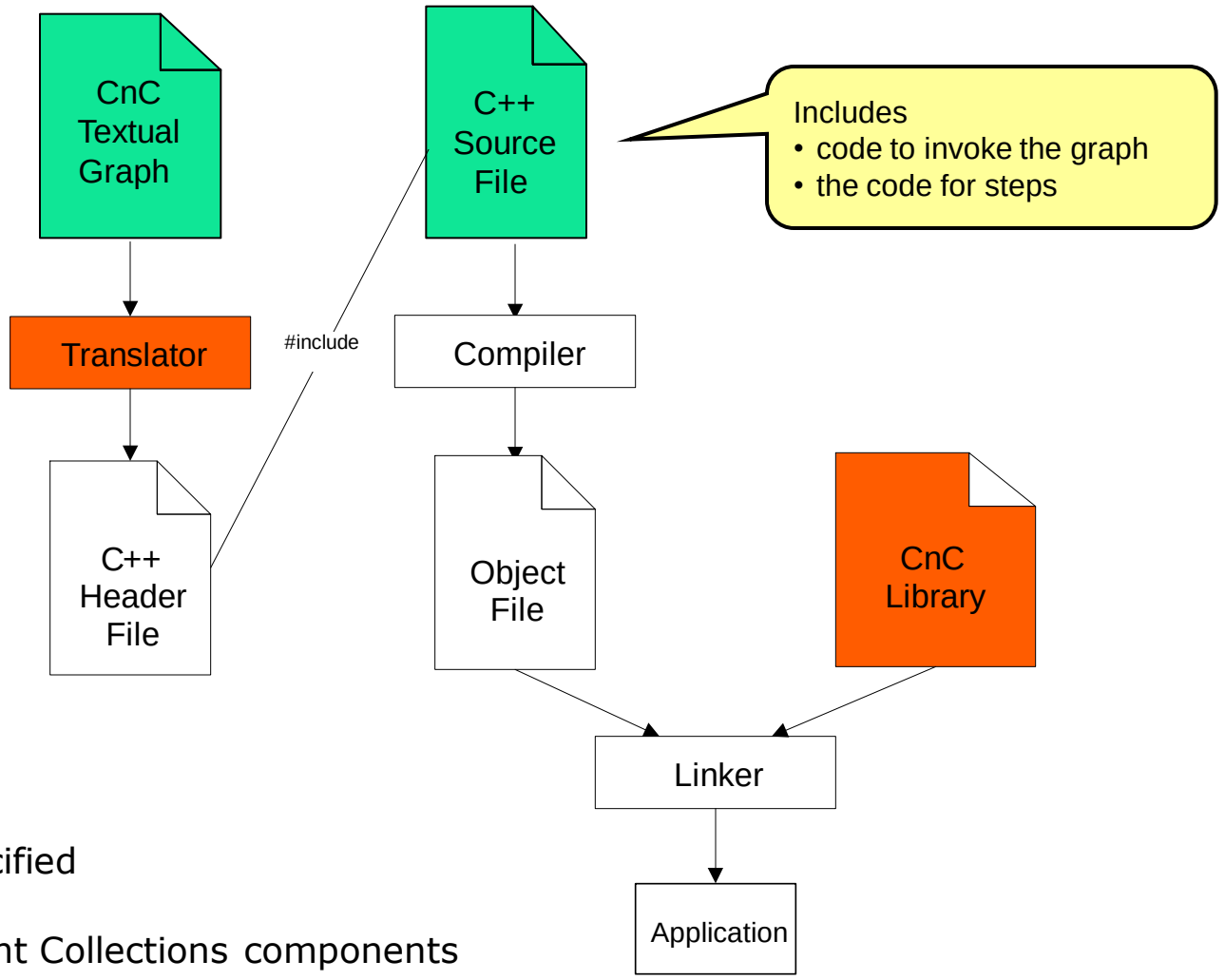
Intel® Concurrent Collections for C++

- Runtime Library
  - For Windows/Linux IA-32 and Intel-64
- Translator
  - A tool to make it easier to use the CnC runtime
  - Command line program
  - Translate `.cnc` file into C++ header file and coding hints file
- Visual Studio 2005 & 2008 IDE integrations
  - Add-ins that set up the CnC paths and register `.cnc` files to be processed by the translator
- Documentation
- Samples

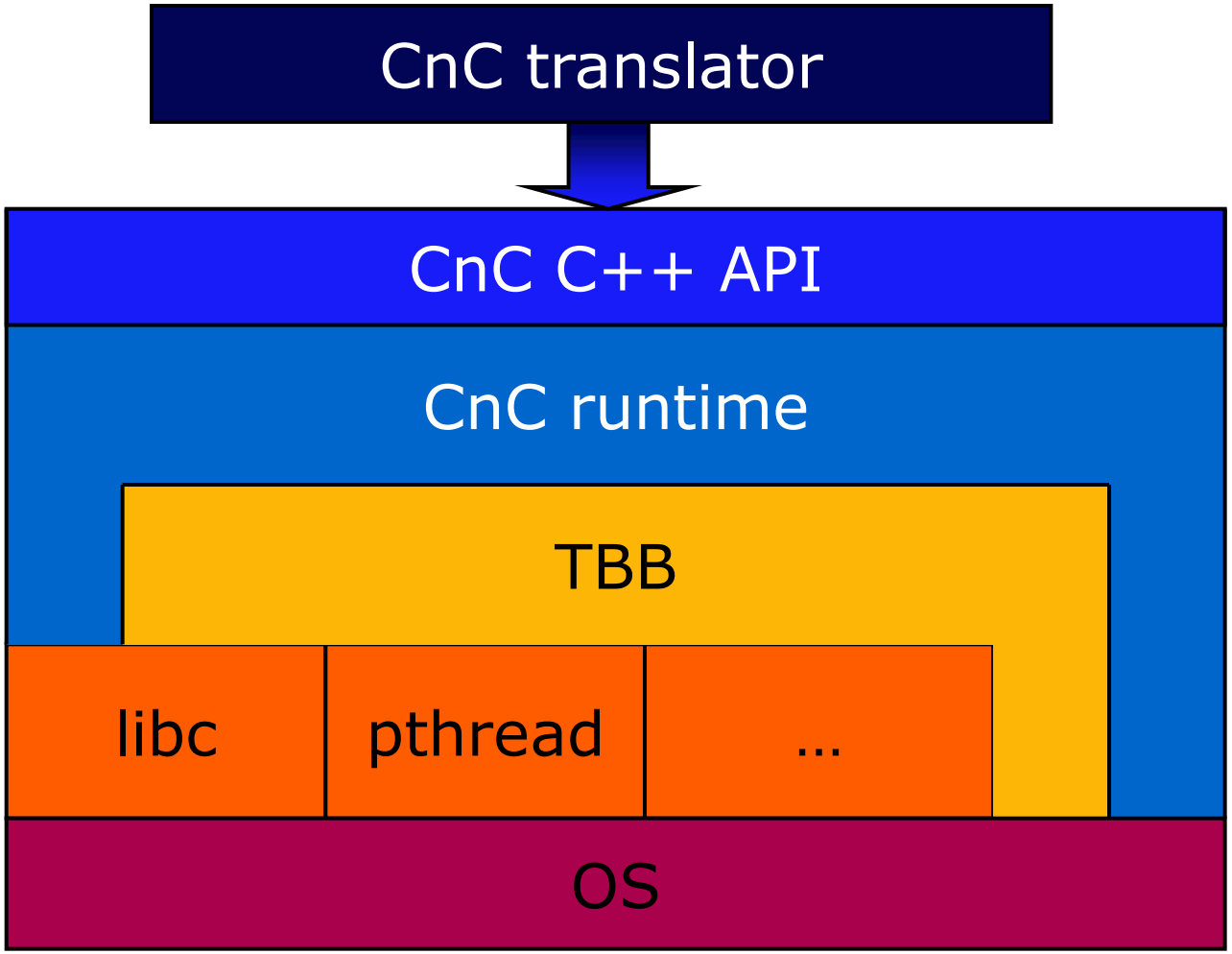
Translator and C++ API

- CnC language and translator can only create skeleton, which uses the API
- C++ coding required in any case
- The API puts CnC concepts into C++
  - Generality through collection being templates
  - Type safety
  - Debug and tuning interface
- Additional features include (new in v0.5)
  - Tuning options
  - Range support for tags
  - Running on distributed systems

Build Model

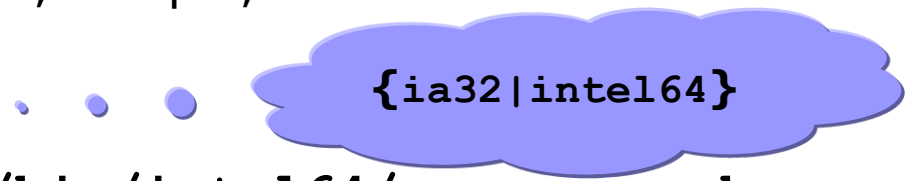


Intel® Concurrent Collections



Linux Installation

- Enter a temporary directory
- **`scp lxbsp1505:/pool/Intel/cnc_0.5_kit.tgz .`**
- Unpack "**`/pool/Intel/cnc_0.5_kit.tgz`**"
- **`cd`** to "**`cnc_0.5_kit`**" and execute install.sh
- Accept license agreement
- Install path could be "…**`/Intel/cnc/0.5`**"
- Ingredients
  - include, doc, sample, misc
  - bin/$arch
  - lib/$arch
- Edit **`<path>/bin/intel64/cncvars.csh`**
  - enter installation path (CNC_INSTALL_DIR)



How to write a CnC application

- Design
  - What are the steps and how are they tagged?
  - When are the tags produced?
  - What are the items, how are they indexed, when are they produced?
- Write textual graph specification in a .cnc file
  - CnC translator generates a header file and a coding-hints file
- Write C++ code of the application matching the graph
  - Include the generated header file
  - Each step is a user function with required interface
  - Steps consume and produce item/tag instances with **`get`** and **`put`**
  - Steps call **`gets`** before any **`puts`** and any memory allocation
  - The environment produces initial tag/item instances and invokes the graph

CnC Translator: Syntax

- Step: name enclosed in parenthesis:
  `(my_step)`
- Item-collection: name enclosed in square brackets:
  `[my_items]`
  - Declaration: leading item type, trailing tag-type:
    `[itemType my_items <itemTagType>]`
- Tag-collection: name enclosed in angle brackets:
  `<my_tags>`
  - Declaration: leading type:
    `<tagType my_tags>`

CnC is a C++ template library.
Tags and items can be of any C++ type.

Major classes provided by CnC C++ API

Tag collection: `CnC::tag_collection< tag_type, range_type >`

`put, put_range, iterate`

Item collection: `CnC::item_collection< tag_type, item_type >`

`put, get, iterate`

Graph/context: `CnC::context< DerivedContext >`

`prescribe`

Context brings everything together

Creating your own context "yourContext"

- Derive from CnC::context< yourContext >
- Declare collections as members
- Initialize collections in context constructor with context as argument
- Prescribe your steps in context constructor

Environment

- Setup TBB:
  `source`
  `/afs/cern.ch/sw/IntelSoftware/linux/x86_64/Compiler/11.1/059/bin/iccvars.csh intel64`
- Recommended to **`source`** **`<cnc-v0.5>/bin/intel64/cncvars.csh`** (requires using **`csh`** or **`tcsh`**)

- Manually, you can do the following
  - Translator and binaries use shared libs
    - **$LD_LIBRARY_PATH** should contain **`<cnc-v0.5>/lib/intel64`**
  - CnC/Intel convention in Makefiles use
    - **$CNC_INSTALL_DIR**
    - **$CNC_ARCH_PLATFORM**
  - Might want to extend **$PATH** to use cnc translator on command line

Tutorial

- Copy tutorial material
  "`scp lxbsp1505:/pool/Intel/cnc_hands_on.tgz`"
- Unpack

- Each exercise comes in a separate directory
- Accompanied by slide-show
- All come with a starting point
  - Initial code (CnC file and/or C++ code)
  - Makefile (link to parent dir's Makefile)
- Exercises start with solution of their predecessors
  - Symbolic link in each directory to solution in successor

Implementing Fibonacci

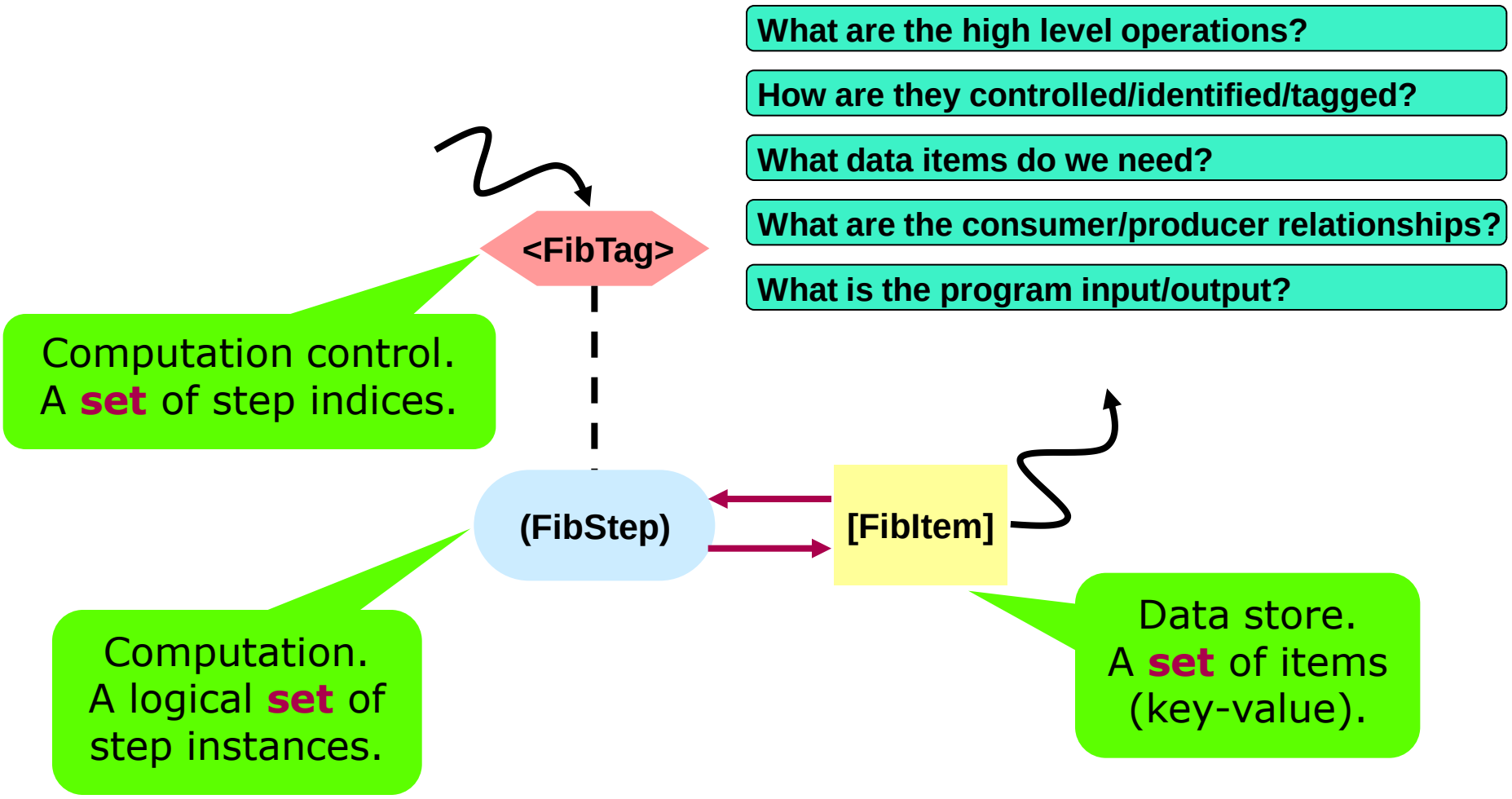
- Start with a CnC-spec
- Use translator to generate
  - header-file and
  - coding hints
- Copy coding hints and edit
- Compile and link executable
- Run!

Fibonacci Numbers

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584

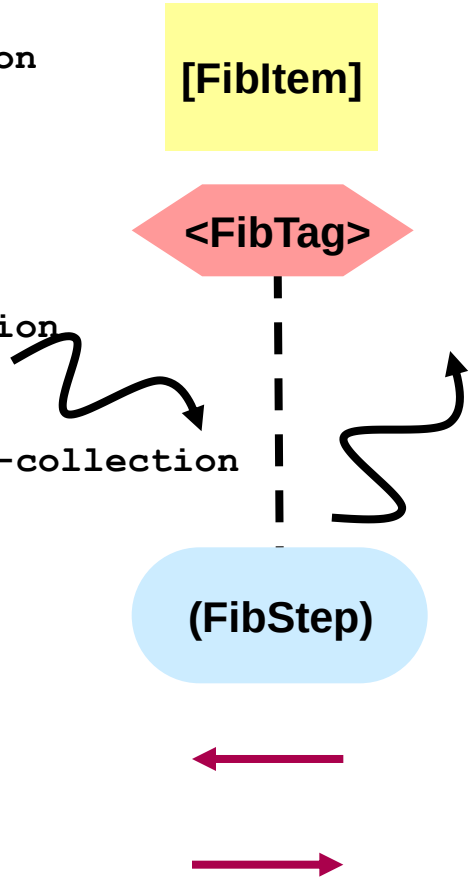
Fibonacci – Graphical CnC



Hands on Fibonacci

Enter directory "`fib`". Edit/create "`fib.cnc`"*

```
// CnC spec to compute fibonacci numbers fib( n ) = fib( n-1 ) + fib( n-2 )

// here we store the fibonacci numbers in an item-collection
[fib_type m_fibs <fib_type>];

// we need a tag-collection to control evaluation
<fib_type m_tags>;

// the environment prescribes tags and so triggers evaluation
env -> <m_tags>;

// the environment reads/consumes the result from the item-collection
env <- [m_fibs];

// prescribe/control step "fib_step"
<m_tags> :: (fib_step);

// the step reads/consumes fibonacci numbers from m_fibs
[m_fibs] -> (fib_step);

// the step also writes numbers
(fib_step) -> [m_fibs];
```



Translate!

```
[fib]> cnc fib.cnc
Or
[fib]> make fib.h
```

Creates fib.h and fib_codinghints.txt

Generated Header (fib.h)

```
// … declarations etc.

struct fib_context : public CnC::context< fib_context >
{
    // Item collections
    CnC::item_collection< fib_type, fib_type > m_fibs;

    // Tag collections
    CnC::tag_collection< fib_type > m_tags;

    // The context class constructor
    fib_context()
        : CnC::context< fib_context >(),
          // Initialize each item collection
          m_fibs( this ),
          // Initialize each tag collection
          m_tags( this, false )
    {
        // Prescriptive relations
        prescribe( m_tags, fib_step() );
    }
};
```

- Step-declarations
- Context definition
  - Collections
- Context constructor
  - Step-prescription

Coding Hints - main

Copy "`fib_codinghints.txt`" to "`fib.cpp`". Edit/create "`fib.cpp`"*

```
typedef unsigned long long fib_type; // define our type
#include <iostream>                  // needed for output
#include "fib.h"

int main( int argc, char * argv[] )
{
    // Create an instance of the context class which defines the graph
    fib_context c;

    fib_type n = atol( argv[1] );   // read from command line, not safe...

    // For each tag value from the environment (ENV), put the tag into
    // the proper tag collection
    for( fib_type m_tags_Tag = 0; m_tags_Tag <= n; ++m_tags_Tag ) // need to iterate
       c.m_tags.put( m_tags_Tag );

    // Wait for all steps to finish
    c.wait();

    // For each output to the environment (ENV), get the item using the
    // proper tag
    fib_type m_fibs_ENV;
    c.m_fibs.get( n, m_fibs_ENV );
    std::cout << "fib(" << n << ") = " << m_fibs_ENV << std::endl; // print

    return 0;
}
```

Step execution might start before `wait()`.

Software

Writing steps

When prescribing a step, the step must be a class with an execute method:

```
class my_step
{
    int execute( const tagType & tag,
                 yourContext & context ) const;
};
```

`execute` must be `const` with no side-effects
The step can have a global/constant status
The step must be copy-constructible

Coding Hints – step code

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

Delete the provided hints in the step-body and add the appropriate code to compute fib.

Compile and link!

At compile time, you need

- `-I$CNC_INSTALL_DIR/include`
- `-I$TBB_INSTALL_DIR/include`

At link time, you need

- `-L$CNC_INSTALL_DIR/lib/$CNC_ARCH_PLATFORM`
- `-lcnc`
- `-L$TBB22_INSTALL_DIR/lib/$TBB_ARCH_PLATFORM`
- `-ltbb -ltbbmalloc`

```
[fib]> make fib
```

Creates binary “fib”.

Run!

At run time, your **LD_LIBRARY_PATH** must include
$CNC_INSTALL_DIR/lib/$CNC_ARCH_PLATFORM

```
[fib]> ./fib 10
```

```
[fib]> env LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:<path>\
       ./fib 10
```

```
[fib]> ./fib 10
fib(10) = 55
```

What we've learned

- CnC is focused on semantics
- Translator generates data structures, interfaces and coding hints and so eases getting started
- How to translate, compile, link and run a CnC program
- No thinking about parallelism

Execution trace

- CnC can print an execution trace to stdout
- Selectively per collection
- Coding hints:

```
// Debug trace can be enabled for a collection with the call:
// CnC::debug::trace( c.m_tags, "m_tags" );
```

In "`fib.cpp:main`": Uncomment trace statement (2nd line)
Build and run

```
[fib]> ./fib 3
Put tag <m_tags: 0>
Put tag <m_tags: 1>
Put tag <m_tags: 2>
Put tag <m_tags: 3>
fib(3) = 2
```

A more interesting execution trace

- Let's trace all collections:

```
CnC::debug::trace( c.m_tags, "m_tags" );
CnC::debug::trace( c.m_fibs, "m_fibs" );
CnC::debug::trace( fib_step(), "fib_step" );
```

In "`fib.cpp:main`":
Insert the two additional trace statements above
Build and run

```
./fib 2
Put tag <m_tags: 0>
Put tag <m_tags: 1>
Put tag <m_tags: 2>
Start step ( fib_step: 2;  )
Start step ( fib_step: 0;  )
Get item [m_fibs: 1]  ( not ready )
Put item [m_fibs: 0] -> 0
Start step ( fib_step: 1;  )
Put item [m_fibs: 1] -> 1
End step ( fib_step: 0 )
End step ( fib_step: 1 )
Requeue step ( fib_step: 2 )  ( input item not ready )
Start step ( fib_step: 2;  )
Get item [m_fibs: 1] -> 1
Get item [m_fibs: 0] -> 0
Put item [m_fibs: 2] -> 1
End step ( fib_step: 2 )
Get item [m_fibs: 2] -> 1
fib(2) = 1
```

- If an item is unavailable, the requesting step is rolled back
- A step will be re-scheduled once the item becomes available

Scheduler Statistics

- CnC can print scheduler statistics when the context is destructed:

```
CnC::debug::collect_scheduler_statistics( c );
```

In "`fib.cpp:main`": Comment out/delete trace statements
Add the above statement
Build and run

```
[fib]> ./fib 10
fib(10) = 55
Steps created( 11 )
Steps scheduled( 20 ) inflight( 0, 0 )
Steps requeued( 9 ) resumed( 9 )
```

What we've learned

- CnC debugging API helps with understanding the runtime behavior

Don’t like the put-loop

- It would be more elegant to invoke step recursion within the step and let the env put only one tag

`Put`ting tags before `get`s can express call/return (or concurrent co-routine).

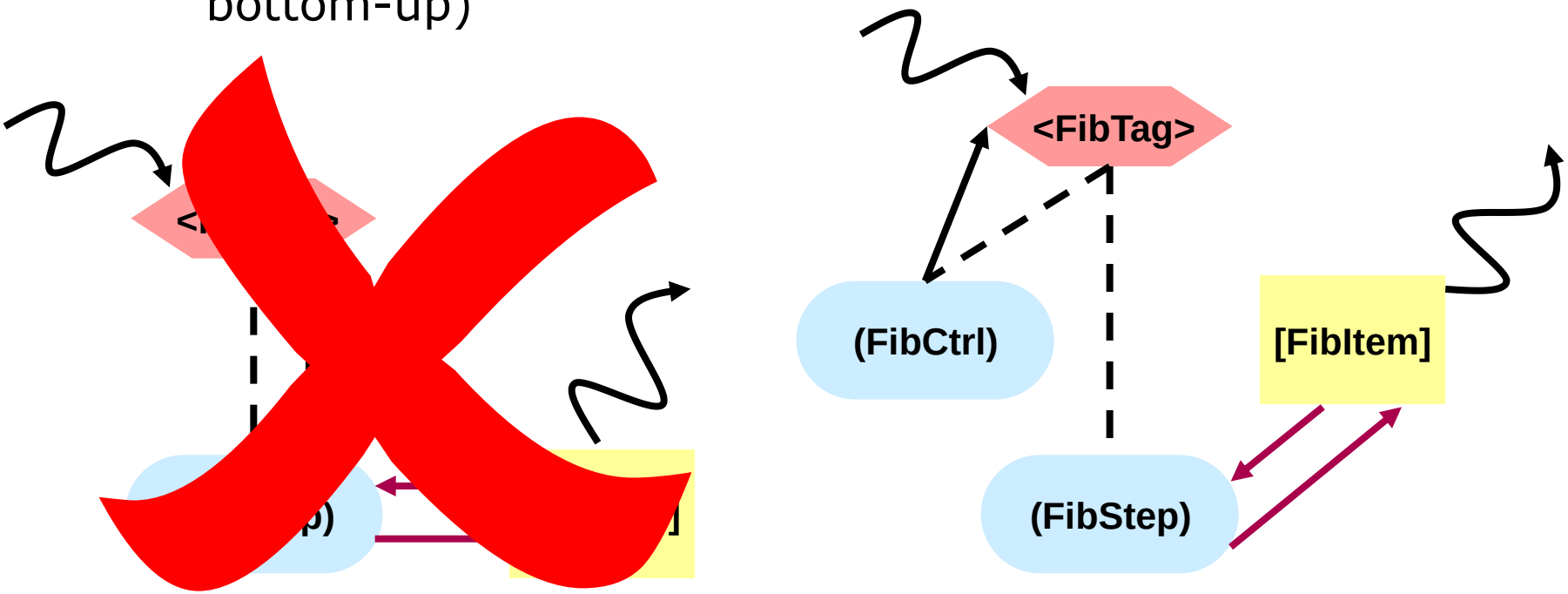
This is not supported by the CnC model.

`Put`ting items before `get`s is not supported by our replay implementation (exception!).

Steps must not put items or tags before getting any items.

Avoiding cycles

- Let's split the step into
  - One step for execution control (controller/controlee, top-down)
  - Another step for actual computation (producer/consumer, bottom-up)



Hands on CnC-spec!

Enter directory "`fib_rec`"
In "`fib.cnc`":
Add the extra step, its prescription dependencies

```
...

// prescribe/control step "fib_ctrl"
<m_tags> :: (fib_ctrl);

// the control step also put tags
(fib_ctrl) -> <m_tags>;
```

Translate
`[fib_rec]> cnc fib.cnc`
Or
`[fib_rec]> make fib.h`

Hands on C++ code!

In "`fib_rec/fib.cpp:main`": Replace for-loop with

```
c.m_tags.put( n );
```

Add new step (can copy signature from coding-hints).

```
int fib_ctrl::execute(const fib_type & t, fib_context & c ) const
{
    if( t > 0 ) c.m_tags.put( t - 1 );
    if( t > 1 ) c.m_tags.put( t - 2 );

    return CnC::CNC_Success;
}
```

Build

```
[fib_rec]> make fib
```

Run!

What we've learned

- Get all items before putting tags or items
- CnC runtime can do the hard part for you

Writing steps safely

Always get all items before putting tags or items

Defer allocation and computation to after last get

- If an item is unavailable, an exception will be thrown.
- The runtime catches the exception and re-starts the step once the item becomes available.
- The above rules protect you from
  - Memory leaks
  - Dead-locks
  - Inefficient code

Tuning

- Intel® Concurrent Collections for C++ comes with tuning knobs
- Programmer gives hints to the runtime
  - Identical tags might be put more than once
    - ⇨ Runtime can perform or avoid redundant step execution. Tradeoff between recording-cost and reduced computation.
  - Pre-declare data dependencies
    - ⇨ Runtime can choose to delay step execution until all items are available
  - Number of gets to items
    - ⇨ Garbage collection: if an item's get-count has reached 0, the runtime can remove it (and free memory)

Multiply Prescribed Tags

- Most tags in our recursive Fibonacci example are prescribed **multiple times** (due to multi-way recursion)
- By default CnC runtime creates a step-instances for each tag

```
[fib]> ./fib 10
fib(10) = 55
Steps created( 11 )
Steps scheduled( 20 ) inflight( 0, 0 )
Steps requeued( 9 ) resumed( 9 )
```

```
[fib_rec]> ./fib 10
fib(10) = 55
Steps created( 464 )
Steps scheduled( 483 ) inflight( 0, 0 )
Steps requeued( 19 ) resumed( 19 )
```

- Runtime can be instructed to preserve tags and so execute a step only once for each tag-value

Preserving Tags

Enter directory "`fib_preserve`"
Int "`fib.cnc`":
Add the `preserve` attribute to the tag-collection.

```
...
// we need a tag-collection to control evaluation
<fib_type m_tags> preserve=true;
...
```

Build
`[fib]> make fib`
Run!

```
[fib_rec]> ./fib 10
fib(10) = 55
Steps created( 464 )
Steps scheduled( 483 ) inflight( 0, 0 )
Steps requeued( 19 ) resumed( 19 )
```

```
[fib_preserve]> ./fib 10
fib(10) = 55
Steps created( 22 )
Steps scheduled( 31 ) inflight( 0, 0 )
Steps requeued( 9 ) resumed( 9 )
```

The Tuner

- Class/object attached to step prescriptions
- Provided by programmer
- Controls tuning knobs through giving hints
  - Pre-declare dependencies to consumed items
  - Declare a step as non thread-safe
  - Provide partitioning strategy for using ranges
  - Provide distribution strategy for distributed CnC
  - others
- Default implementation

Providing a tuner

Enter directory "`fib_tuner`"
In "`fib.cnc`":
Add a (new) step-declaration with `tuner` attribute.

```
...
(fib_step) tuner=fib_tuner;
...
```

Providing a tuner

In "`fib_tuner/fib.cpp`": before #including "`fib.h`"
declare `fib_context`
add `fib_tuner`
Might copy it from "`fib_tuner/fib_codinghints.txt`"
Build and Run!

```
typedef unsigned long long fib_type; // define our type
#include <iostream>                  // for printing to stdout
#include <cnc/cnc.h>

struct fib_context;

struct fib_tuner : public CnC::default_tuner< fib_type, fib_context >
{ // empty class for now, using defaults
};

#include "fib.h"
...
```

Tuner is empty
✓No change in runtime behavior yet

Filling up the tuner

Provide "depends" method and declare consumed items. Build and Run!

```
...
struct fib_tuner : public CnC::default_tuner< fib_type, fib_context >
{
    template< typename T >
    void depends( const fib_type & tag, fib_context & c, T &dC ) const;
};

#include "fib.h"

template< typename T >
void depends( const fib_type & tag, fib_context & c, T & dC ) const
{
    if( tag > 1 ) {
        dC.depends( c.m_fibs, tag - 1 ); // step will consume this
        dC.depends( c.m_fibs, tag - 2 ); // step will consume this
    }
}
...
```

```
[fib_tuner]> ./fib 10
fib(10) = 55
Steps created( 22 )
Steps scheduled( 22 ) inflight( 0, 0 )
Steps requeued( 18 ) resumed( 18 )
```

Runtime Options

- CnC runtime has two switches used through environment variables
  - CNC_NUM_THREADS sets the number of threads
  - CNC_SCHEDULER sets the scheduling policy
    - TBB_TASK (default): TBB's task-stealing
    - TBB_QUEUE: custom task queue
      - CNC_STEAL=1 with local queues and task-stealing
      - CNC_STEAL=0 one global task queue
    - TBB_WHILE: TBB's task-stealing through a while-loop
    - TBB_FOR: TBB's task-stealing through a for-loop and a double buffer
    - TBB_QUEUE_ONLY_STEALING: custom task stealing with local queues only

Run with CNC_SCHEDULER set to TBB_QUEUE

What we've learned

- CnC runtime is tunable
- Translator greatly helps with getting started
- Later in the development cycle it might add overhead

Garbage Collection

- Dead items can be removed and memory freed
- Current CnC cannot detect when an item becomes dead

⇨Programmer can provide the number of expected gets when an item is put

Providing get-counts is not always possible.

It breaks if steps are fully executed more than once.

Providing get-counts

Enter directory "`fib_gc`"
in "`fib.cpp`":
- **Enable tracing of items (`m_fibs`)**
- **Add expected get-count when `put`ting items.**

Build and Run!

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

```
[fib_gc]> ./fib 10
Put item [m_fibs: 1] -> 1 getcount=2
Put item [m_fibs: 0] -> 0 getcount=1
Get item [m_fibs: 1] -> 1
Get item [m_fibs: 0] -> 0
Put item [m_fibs: 2] -> 1 getcount=2
item [m_fibs: <1>] m_getCount decremented to 1
item [m_fibs: <0>] m_getCount decremented to 0
Get item [m_fibs: 2] -> 1
Get item [m_fibs: 1] -> 1
Put item [m_fibs: 3] -> 2 getcount=2
item [m_fibs: <2>] m_getCount decremented to 1
item [m_fibs: <1>] m_getCount decremented to 0
Get item [m_fibs: 3] -> 2
Get item [m_fibs: 2] -> 1
Put item [m_fibs: 4] -> 3 getcount=2
item [m_fibs: <3>] m_getCount decremented to 1
Get item [m_fibs: 4] -> 3
item [m_fibs: <2>] m_getCount decremented to 0
Get item [m_fibs: 3] -> 2
Put item [m_fibs: 5] -> 5 getcount=2
item [m_fibs: <4>] m_getCount decremented to 1
Get item [m_fibs: 5] -> 5
item [m_fibs: <3>] m_getCount decremented to 0
```

What we've learned

- Garbage collection requires understanding of the dynamic graph execution

Distributed CnC

- Current release V0.5 on whatif.intel comes with support for distributed memory
- Runtime can offload step execution to other machines
- If standard data types are used, only minor changes are needed to existing CnC code
- Serialization for non-standard data types is simple. Only one method/function for marshalling and un-marshalling needed.
- Work distribution can be controlled through tuner

Distributing Fibonacci

Enter directory "`fib_dist`"
In "`fib.cpp`":
`#include <cnc/dist_cnc.h>`
in `main`: add context-initializer
Build (`[]> make fib DIST=1`) and Run!

```
...
#ifdef _DIST_
#include <cnc/dist_cnc.h>
#else
#include <cnc/cnc.h>
#endif
...
int main( int argc, char * argv[] )
{
#ifdef _DIST_
    CnC::dist_cnc_init< fib_context > _dinit;
#endif
...
```

Manually Going Distributed

```
[fib_dist]> ./fib 10
start clients manually with contact string: 0:1025_111@127.0.1.1
```

In a separate shell

- **Log into another machine (if possible)**
- **Enter directory "`fib_dist`"**
- **Run `./fib` with environment-var**
  CNC_SOCKET_CLIENT set to contact string

```
[fib_dist]> env CNC_SOCKET_CLIENT=0:1025_111@127.0.1.1 ./fib
```

```
fib(10) = 55
Steps created( 12 )
Steps scheduled( 11 ) inflight( 0, 0 )
Steps requeued( 18 ) resumed( 18 )
```

Going Distributed with a script

- If environment variable CNC_SOCKET_HOST is set to a number, the host expects as many clients
- Otherwise it is interpreted as a name of a dual-mode script which
  - either returns number of clients
  - or launches them
- CnC kit comes with a sample script "**`misc/start.sh`**"
  - Launches clients on **`localhost`**, but using **`ssh`**
  - Copy and edit the script to start clients on remote hosts (just replace "**`localhost`**" with the respective names) and/or to adapt the number of clients

Going Distributed with a script

Run `./fib` with environment-var
CNC_SOCKET_HOST set to the launch-script

```
[]> env CNC_SOCKET_HOST=$CNC_INSTALL_DIR/misc/distributed/socket/start.sh \
? `pwd`/fib 10
--> established socket connection 3, 2 still missing ...
--> established socket connection 2, 1 still missing ...
--> established all socket connections to the host.
--> establishing client connections to client 1 ... done
--> establishing client connections to client 2 ... done
fib(10) = 55
Steps created( 11 )
Steps scheduled( 9 ) inflight( 0, 0 )
Steps requeued( 13 ) resumed( 13 )
```

Things to keep in mind

- Collections must be members of contexts (constructed in its ctor)
- Context must be default constructible and prescribe steps there
- Pointers are dangerous
  - Tags must not be of pointer type
  - Items of pointer type need special treatment; better avoid them
- Global variables are evil and must not be used (within the execution scope of steps)
- In contrast to local-only execution, preservation of tags will only locally suppress redundant step execution.

- **All this is aligned with CnC's methodology!**

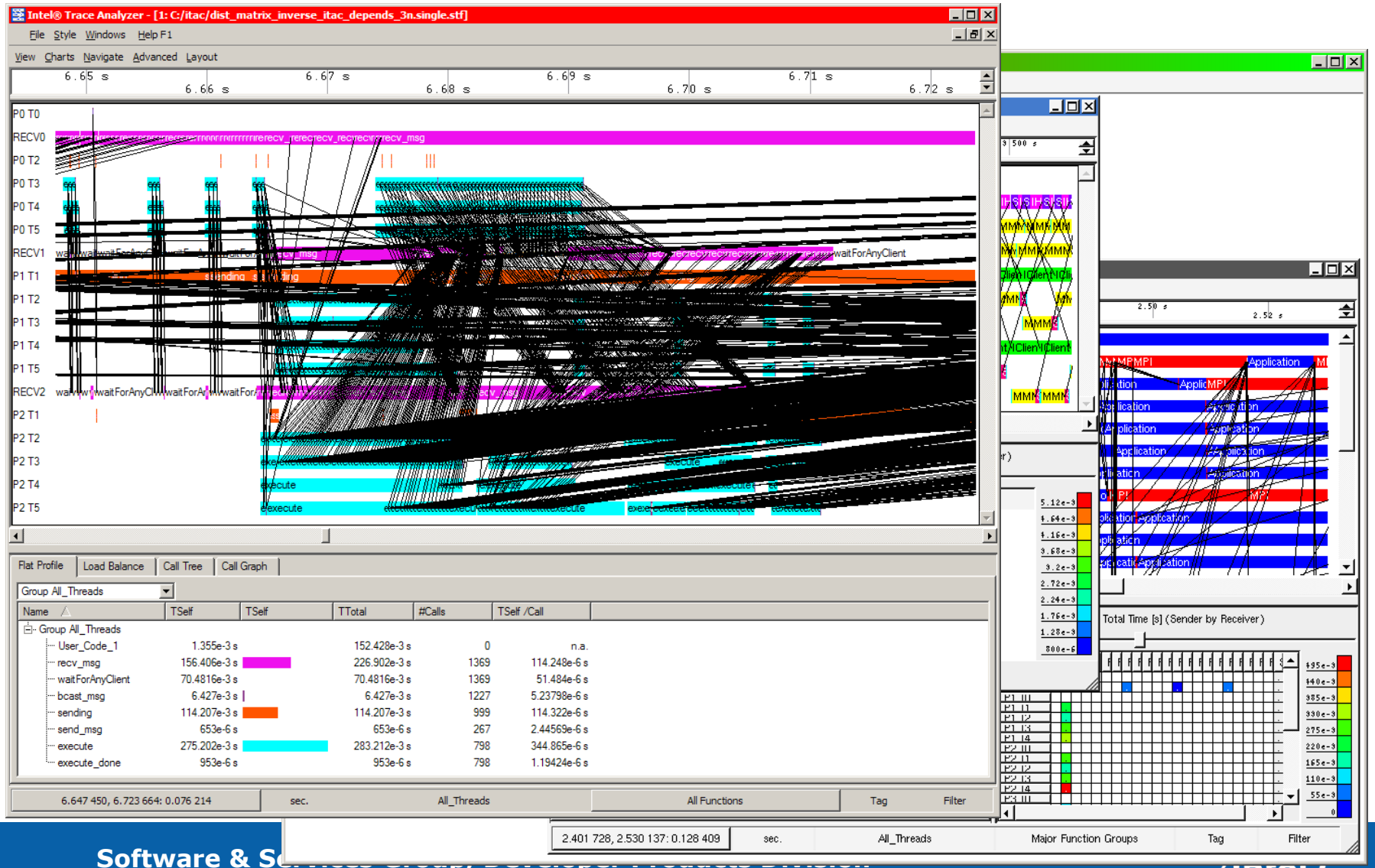
Serialization Example

```
struct my_struct
{
        bool     m_isPartitioned;
        bool     m_verbose;
        int      m_size;
        double * m_array;

        void serialize( CnC::serializer & ser )
        {
                ser & m_isPartitioned & m_size & m_verbose;
                ser & CnC::array_alloc( m_array, m_size );
        }
};
```

Alternatively a (global) serialize function can be used. Simple structs/classes are serialized trivially through

```
CNC_BITWISE_SERIALIZABLE( type );
```

Debugging and Profiling



What we've learned

- With some discipline, almost any valid CnC program can easily be ported to run across a network – with little effort.

Tag Ranges

- Step execution comes with scheduling overhead
- When putting single tags, the runtime can't (easily) agglomerate step-instances
- The CnC C++ API provides means to put entire ranges of tags (in the spirit of TBB ranges)
- The runtime can then partition/split the range and schedule several steps at once
- For small steps, ranges can lead to significant performance gains

Tag-Ranges do not play well with the translator (yet).

Range Features

- Tags can have any type
- Works with **`TBB::blocked_range<…>`**
- Custom ranges possible, narrow interface required
  - Split-constructor
  - Iterator
- Comes with
  - fixed-size parititioner
  - Adaptive-size partitioner (depends on # threads)
  - Partitioning of self-dividing tags/ranges
- Allows custom partitioners (also with a narrow interface)

Primes

- Basic implementation puts many individual tags in a loop
- This could be replaced by a range
- tbb::blocked_range does not support strides, so we will use CnC's internal range
- Using ranges is expected to decrease
  - Number of scheduled steps
  - runtime

Enter directory "`primes_ranges`"
Let's start with running basic version for comparison.
Build (`make primes`) and Run (`./primes 646678`)!

Hands on Ranges

Enter directory "`primes_ranges`"
In "`primes.cpp`": declare range type with tag-collection
replace loop with putting a range
Build (`make primes`) and Run (`./primes 646678`)!

```
...
struct my_context : public CnC::context< my_context >
{
  CnC::tag_collection< int, CnC::Internal::strided_range< int > > m_tags;
  CnC::item_collection< int,int > m_primes;
...
```

```
...
  tbb::tick_count t0 = tbb::tick_count::now();

  if( n > 2 )
    c.m_tags.put_range( CnC::Internal::strided_range< int >( 3, n+1, 2 ) );

    c.wait();
...
```

Comparison

```
[primes_basic]> ./primes 646678
Determining primes from 1-646678
Found 52571 primes in 4.4708 seconds
Steps created( 323338 )
Steps scheduled( 323338 ) inflight( 0, 0 )
Steps requeued( 0 ) resumed( 0 )
```

```
[primes_range]> ./primes 646678
Determining primes from 1-646678
Found 52571 primes in 5.3769 seconds
Steps created( 43 )
Steps scheduled( 42 ) inflight( 0, 0 )
Steps requeued( 0 ) resumed( 0 )
```

Load Balancing issue!
We'd need a more sophisticated partitioner.

Left to the interested as an exercise…

Intel® Concurrent Collections

http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/

Contact: kath.knobe@intel.com
frank.schlimbach@intel.com
mario.deilmann@intel.com

Sponsored by Intel/SSG/DPD
Technology, Pathfinding & Innovation (Geoff Lowney)
& ICL/Languages and IDEs (I-Yu Chen)

Kudos to Melanie Blower, Chih-Ping Chen and Shin Lee for their contribution to the training material.

The End.

Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

http://intel.com/software/products

66