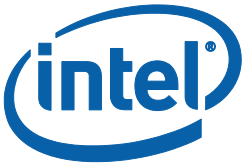
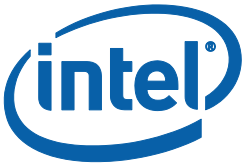


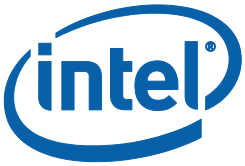
Intel® Concurrent Collections for C++

Tutorial

World Wide Web: http://www.intel.com

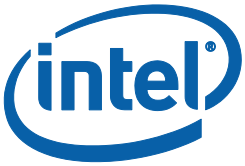


CONTENTS



CNC C++ TUTORIAL

The CnC programming model is quite different from most other parallel programming models in several important ways. Hence, we highly recommend you carefully read the following introduction sections (CnC in a Nutshell) carefully before diving into the hands-on tutorial. It'll take only a few minutes but you'll have a much easier time, more fun and faster success! Promise, it'll pay off!



CNC IN A NUTSHELL

CnC is a new programming model which is different from most others. It designed for creating parallel applications, but not for expressing parallelism explicitly. In CnC the programmer declaratively specifies the dependencies among computation units, but does not in any way indicate how those are to be met. It is specifically designed for addressing the coordination among potentially parallel computation units and data.

In essence, the programmer declares certain dependencies between two (or more) computation units. Specifying the dependencies is simpler than expressing parallelism, because it only makes application semantics explicit - and does not depend on the platform or any specific parallelization technique. Additionally, it exposes more parallelism potential because it does not bind a particular parallelism to the algorithm/program. The CnC runtime will do the hard work of figuring out how to execute things in parallel; it will try to maximize parallelism, limited only by the ordering constraints defined by the programmer.

The only 2 ordering constraints which semantically exist in any program come from the following 2 relationships

1. producer/consumer
2. controller/controlee

It is apparent that a producer needs to executed before the consumer can run. Similarly, if one computation unit decides whether another computation needs to be executed or not, the decision maker (e.g. the controller) needs to go before the controlled computation(s).

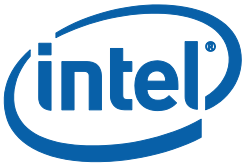
These two relationships (producer/consumer and controller/controlee) are the only relations needed to determine semantically correct parallel or sequential execution orders. This information is known to any programmer, even when writing sequential programs. It only gets lost because traditional programming languages have no means to explicitly express it. Exactly this is what a CnC program specifies (on top of the computation functionality of course).

BURGERS WITH FRIES AND PIES FOR DESSERT

BURGERS AND FRIES ARE PARALLEL

Let's make burgers and French Fries. Let's say the ready-to-process ingredients (cut potatoes and prepared meat) get delivered to a service hatch, from which the cook can take them for processing. In CnC such hatches are *item-collections* , which store input and output data/items. Each collection can hold multiple instances of the same data(-type). In our kitchen that would mean we have a hatch for potatoes and a another hatch for the meat. Similarly, whatever we create and cook (our output) would be placed in such a hatch (item-collection). Our hungry guests can pick up the delicacies from this hatch once they have been produced.

The tasks for our cooks are frying the potatoes and barbecuing the meat. In CnC we call such tasks *steps* , which are basically just normal functions. The interesting question is: are there any dependencies between the two cooking tasks/steps? Of course not: neither uses the result of the other and neither decides whether the other needs to be done (or not). Hence, there is no ordering required, we can fry the potatoes first, or do the meat first or do them in parallel.



NOTE:

If we insisted on working one-handed and allowed only one cook active at the same time, we needed to decide on a (arbitrary) sequence. In serial languages, the programmer is required to make that arbitrary choice but doesn't indicate that the choice was optional.

CAN'T BAKE A PIE UNTIL IT'S PREPARED

Let's add mini cherry pies to our meal for dessert. A mini-pie has to be prepared and baked (prepare_pie, bake_pie). Altogether, we now have four steps, three of which (barbecue_burger, fry_potatoes, prepare_pie) have no dependencies between them. However, we need the pie in order to bake it. We don't need to know where it comes from, we only need the ready assembled pie: we take it from the hatch once it's there and bake it. Similarly, for the task of preparing a pie we don't need to know what's going to happen with it afterwards, we just create it and put it on the hatch.

Incidentally, there is a *producer/consumer* relationship between prepare_pie and bake_pie. A producer/consumer relationship constitutes one of the two before-mentioned ordering constraints: the producer needs to be executed before the consumer can use the produced item. Note, the programmer does not explcitily indicate a specific execution order. The runtime takes care of it: as it knows about the relationship, it will never bake a pie before it has been prepared.

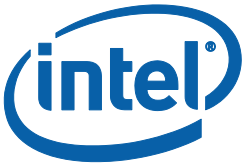
WAITER CONTROLLING COOKS

Now, let's say our guest are individuals and not all of them want the same menu. Some of them might want the full menu: a burger, fries and a pie. Someone else might prefer 2 burgers, no fries but a pie, and so on. So let's take orders (take_orders) first. Only if they know the orders, our cooks can actually start cooking. When taking orders, we must assign a unique identifier (a tag) to each order so that we later know which burgers, fries and pies are for which guest. To communicate the orders with the kitchen, our CnC kitchen uses 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 simply puts the corresponding guest-tag into the step's bowl. For example, if guestA wants a burger and Fries, we put the tag/token 'A' into barbecue_burger's bowl and into fry_potatoes' bowl. Our cooks will then prepare a burger and fries for guest 'A', but as there is no token/tag in the pie bowl, they will not make a pie for 'A'. CnC calls such bowls *tag-collections* .

We have just declared a controller/controlee relationship. Our waiter decides which steps need to be executed. Moreover, our CnC kitchen assigns a tag (a unique identifier) to each execution instance. With this our chef (e.g. the CnC runtime) can now coordinate the different tasks as they come in (e.g. to make sure burgers and fries from the same order are ready together).

WE HAVE A CNC SPECIFICATION!

We have now introduced the dependencies which imply a certain (partial) ordering. This partial ordering allows us (e.g. the CnC runtime) to determine a legal scheduling of the step instances (computation units), be it serial or



parallel. If we have ten cooks or one, all they need to know are the ordering constraints (besides knowing how to cook, of course). They could do everything in parallel, except they need to

1. wait for the tags to come in before starting a task
2. wait for each pie to be prepared before baking it.

NOTE:

> The constraints we defined are semantic attributes, they are requirements only of the algorithm. They exist independent of the programming language or programming tool, e.g. even if not using CnC they must be obeyed: a pie simply needs to be prepared before put into the oven. And it doesn't make sense to start frying potatoes unless we know that someone actually wants fries.

Basically we came up with a CnC specification for our kitchen problem. What we have done is

- identified the computation units (barbecue_burger, fry_potatoes, prepare_pie, bake_pie and take_orders)
- identified the data entities (meat, potatoes, fries, burgers, pies)
- identified the control tags (guest-id)
- consumer/producer relationships (prepare_pie -> pies -> bake_pie)
- controller/controlee relationships (take_orders => xxx_bowls => steps)

That's what makes a CnC specification; paired with an implementation of each computation step it makes an application ready for parallel execution. Nothing else is needed in the CnC world - the rest is handled by the CnC runtime. The CnC approach leaves the CnC runtime with all freedom to execute things in parallel as long as it satisfies the defined semantics.

BUT...

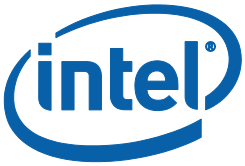
..of course there are rules when programming in CnC:

1. Computation units (steps) must execute statelessly, e.g. computation must not access or even alter any global data and leave no traces behind other than putting things into CnC's collections.
2. Data is immutable. Once put, data-items cannot be altered. instead of changing a value, in CnC you put a new value with a new tag.

The rules might sound more restricting than they actually are. Actually, in this tutorial you will almost certainly start appreciating the rules. They are much easier to follow than the common rules of traditional parallel frameworks. For example, almost all threading frameworks tell the programmer "Don't create races". That's a very vague rule and verifying it's correct implementation is close to impossible. CnC's rules are easy to follow, easy to check and they will actually give race-freedom and determinism without you needing to think about it.

THE CNC RECIPE

The challenge in using CnC certainly lies in applying its concepts to a given problem (application). Luckily, once that's accomplished, the remaining tasks are straight forward and later changes to the design are relatively simple to make. The thought process to getting to a complete CnC design for your application can happen in several small phases:

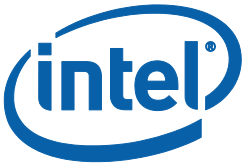


1. Define the data and computation entities of your application (step-, item- and tag-collections)
2. Define how to distinguish between instances of these entities (what do the identifiers/tags look like?)
3. Define the relations between the entities (producer/consumer and controller/controlee)

This tutorial will guide you through these phases with a simple example program. It'll explain the use of CnC with the C++ API by creating a program to compute Fibonacci numbers. Due to nature of Fibonacci CnC doesn't necessarily provide the most natural expressiveness for it. However, Fibonacci has the right complexity and characteristics to demonstrate CnC and Intel's C++ API without getting lost in details of non-CnC related issues.

After creating a first program, we will introduce a debugging-interface and more advanced features like a tuning interface providing powerful tuning knobs.

I am ready, let's go!



GETTING STARTED WITH FIBONACCI

We use Fibonacci as an educative example; what we show here is not meant to be an ideal or elegant implementation of Fibonacci. We only use Fibonacci as a simple enough problem to demonstrate CnC and highlight some of its features.

Let's start.

The Fibonacci number of a given value n is recursively defined as fib(n) = fib(n-1) + fib(n-2). As the values can grow very large, let's define our own data type that we can adjust if needed:

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

THE CNC SPECIFICATION

IDENTIFYING COMPUTATION UNITS

Fibonacci is a very simple algorithm. There only one computation which simply adds the values of the two previous Fibonacci numbers. Let's call it "fib_step". In CnC all execution instances of such a function are held in one collection, a CnC::step_collection. Declaring such a step-collection instance (with name m_steps) for steps of type "fib_step" is as simple as this:

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

As we do not need any other function to compute Fibonacci, one step-collection is enough. More complex programs will of course use more than just a single step-collection.

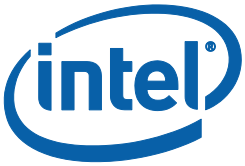
DATA ENTITIES

The only data we seem to care about are the Fibonacci numbers that we compute. The only way in CnC to read values from a computation is through item-collections. So we clearly need to store the final result in such an item-collection. Additionally we need to get the result of another computation to compute a new value. Luckily, Fibonacci is recursive and the kind of the input data is the same as the output data (and with kind we mean that it not only has the same data-type but it also has the same meaning). So we can use one item-collection for the intermediate results and for the final result. More complex programs will of course use more than just a single item-collection.

Defining an item-collection is straight forward:

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

Again, the C++ API provides a template class to define item-collections (CnC::item_collection). The second type-argument is the type of the data to be stored. For our Fibonacci numbers we declared our own type "fib_type", so "fib_type" is the second argument. But what about the first? The first type argument is the tag-type for identifying each data-instance. Just like a traditional key/value pair, the value of our item is accessible (only) through its identifier, the tag. What would make a good identifier for a Fibonacci number? Of course we can simply use an integer; so semantically, the above item-collection maps an integer key to its Fibonacci value. Our "fib_step" will fill this data structure during execution.



In general you can use any C++ type or class for tags and items as long as they provide a copy-constructor (which is true for most types). The tag types must also separately be supported by a hash-function (by default cnc_tag_hash_compare).

STEPS: THE COMPUTATION UNITS

Apparently we need the step which actually computes the Fibonacci number from a given value. Computation units in CnC are defined as steps. Such a step is a class with an execute method which accepts two arguments: a control tag and a second argument, which usually is the context (see below) but could also be anything else.

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

Note that the execute method must be "const" and is not allowed to have side-effects (other than on item-collections).

From the step's perspective, the control tag distinguishes between execution instances of the same step. For example the control tag tells the above fib_step for which Fibonacci number it is currently executed; this should not be confused with input data, which is handled by item-collections.

It is recommended to pass the tag by const-reference, in particular if you are not using standard data types.

CONTROL TAGS

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 will make sure that the step gets executed eventually. So putting a tag tells the CnC runtime that a step-instance needs to be executed, but does not (and can not) say when it is going to run. So let's define the tag-collection which we will use to control our step-collections above:

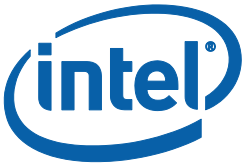
```
    CnC::tag_collection< int >              m_tags;
```

The CnC C++ API once achieves generality through templates; tags can be of any type, as long as it provides a copy-constructor.

THE CONTEXT: BRINGING IT ALL TOGETHER

Before we can actually write the step-code, we need the before mentioned context. The context is what in the CnC literature is often referred to as "graph". It brings together the different collections (tags, items and steps) by defining them as members. Internally it takes care for the runtime mechanics and is used to control the graph evaluation (e.g. waiting for completion). Hence it is recommended to define tag- and item-collections as members of the context.

Each context must be derived from a base class, which again is a template. The accepted template argument is the newly defined derived class. Don't bother about the recursive nature if you are not used to it, it's legal C++. In our case it could look like this



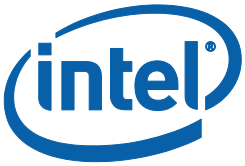
```
struct fib_context : public CnC::context< fib_context > // derive from CnC::context
{
    // the step collection for the instances of the compute-kernel
    CnC::step_collection< fib_step >       m_steps;
    // item collection holding the fib number(s)
    CnC::item_collection< int, fib_type > 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

    // constructor
    fib_context();
};
```

So far we have all the definitions of all collections and the context. Now we need the mechanics, e.g. the relations between the different collections. Producer and consumer relations are declared by calling the respective produces/consumes methods. Then we want that for each tag which is put into the tag-collection m_tags a step from m_steps is executed. We do this by simply calling prescribe on m_tags. We declare all these relations in the context-constructor:

```
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 );
}
```

WRITING THE STEP

Now we have set up the graph and are ready to define the step functionality. The second argument to our step is the context, through which we have access to the tag- and item-collections. To compute the result for a given value, we need the results for the previous two values, which is expressed through getting them from the item-collection. Publishing/producing the result is the reverse operation: a put to the item-collection. The step-code could look as follows:



```
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;
}
```

NOTE:

A step does not care about where the input data comes from, nor does it care about were the output goes to. It only requests/gets the exact data instances it needs and puts the ones it produces. The CnC runtime will take care of all the coordination between producers and consumers. The programmer does not need to think about it (not even if it is run on distributed memory).

WRITING MAIN, THE ENVIRONMENT

Let's now complete the program by providing the "main", in which we instantiate our context

```
    fib_context ctxt;
```

trigger the Fibonacci evaluation

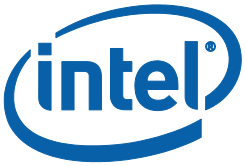
```
    for( int i = 0; i <= n; ++i ) ctxt.m_tags.put( i );
```

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.

Now we wait for the evaluation to complete

```
    ctxt.wait();
```

. The full main could read the desired input value from the command line and print it to stdout:

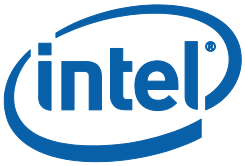


```
int main( int argc, char* argv[] )
{
    int n = 42;
    // eval command line args
    if( argc < 2 ) {
        std::cerr << "usage: " << argv[0] << " n\nUsing default value " << n << std::endl;
    } else n = atol( argv[1] );

    // 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;
}
```

Here is the full example code: fibTutorial.cpp

The code can also be found in the "Samples" directory, including project files for VS on MS Windows* and Makefiles for Linux.

Next: Debugging features



DEBUGGING FEATURES

TRACING

As quite a lot is ongoing under CnC hood, you might sometimes be interested in what's actually happening. The C++ API provides a convenient interface to provide debugging output which describes what's happening. You can control what parts to see and which to be quiet about.

You need to include

```
#include <cnc/cnc_debug.h>
```

which declares the debugging interface. Debug output can be retrieved for steps, tag-collections and item-collections by calling CnC::debug::trace with the respective collections. To trace our Fibonacci step and the item-collection you could issue the respective calls within the constructor of the context or just after creating the context:

```
        // enable debug output for steps
    CnC::debug::trace( ctxt.m_steps );
    // also enable debug output for our items
        CnC::debug::trace( ctxt.m_fibs );
```

when running the program you will see a trace event for every step invocation and for every put/get of items. The trace includes annotations about the successful or unsuccessful completion of the traced operations.

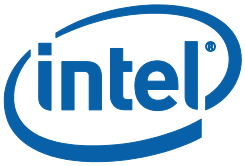
If your appliction uses more than one step-, tag- or item-collection it will be hard to determine to which collection instance individual entries of the default trace belong. To create a more meaningful trace you can assign names to the individual collections. For this their constructor accepts and optional string argument:

```
    fib_context()
        : CnC::context< fib_context >(),
          // Initialize each step collection
          m_steps( *this, "fib_step" ),
          // Initialize each tag collection
          m_fibs( *this, "fibs" ),
          // Initialize each item collection
          m_tags( *this, "tags" )
```

Here is the full example code: fib.h and fib_trace.cpp

SCHEDULING STATISTICS

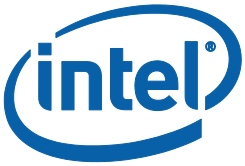
Another interesting feature provides statistics about the internal scheduler. Depending on the availability of dependent items, a step might have been executed to early and needs being replayed when the missing item becomes available. The respective information can be printed when the context is destroyed (in our example when the program terminates). All you need to do is enabling this feature using CnC::debug::collect_scheduler_statistics:



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

In the example fib_stats.cpp we have not placed it into the context's constructor (fib.h), but we could have done so. Of course, it is possible to enable tracing and statistics gathering concurrently.

Next: The Tuners



THE TUNERS

The CnC concepts allow tuning the program independently of the actual program core. Without touching the code in steps and only adding a few declarations in the collection definitions the tuning interface allows providing tuning hints. In the following you will learn about the features of the modular, flexible and easy-to-use tuning interface.

PRE-DECLARING DATA DEPENDENCIES

Supporting the full generality of the CnC programming model obviously comes at some cost in the runtime. In order to accelerate the evaluation of a specific application, the API provides capabilities to influence the execution performance. Most importantly it allows the specification of a "tuner" for each collection. Through the tuners you can provide various hints to the CnC runtime. The tuner-type is an optional template argument to the collection classes. Here is an example of how the tuner type "fib_tuner" is assigned to the step-collection:

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

To define a tuner, you should derive your tuner-class from the default implementations which are provided by the CnC runtime. Otherwise you will need to provide the entire tuner interface even if you intend to use only parts of the interface. The appropriate class for a step-collection tuner is CNC::step_tuner<>:

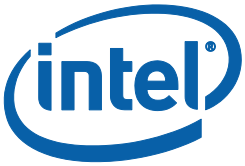
```
struct fib_tuner : public CnC::step_tuner<>
```

The funky "<>" stems from the way C++ handles default template paramemters. This default parameter is only needed for the more advanced use of ranges (Tuning Ranges).

Besides feautures for distributed memory (Running CnC applications on distributed memory) probably the most relevant feature of a step-tuner is the ability to pre-declare item-dependencies. Upon putting a tag normal CnC execution will create a step-instance and hand it over to the underlying scheduler which will eventually launch the step. Only during the actual execution the runtime will observe the unavailability of items (e.g. if a get fails) and needs to re-schedule the step. If the tuner pre-declares items the step-instances will not be scheduled before those items are actually available. To pre-declare item-dependencies you need to provide a template method named "depends", which accepts the same arguments as the step::execute method and an additional parameter. Here's how this would look like in our Fibonacci example:

```
struct fib_tuner : public CnC::step_tuner<>
{
    template< class dependency_consumer >
    void depends( const int & tag, fib_context & c, dependency_consumer & dC ) const;
};
```

Declaring dependencies is straight-forward through calling "depends" on the provided template object.



```
template< class dependency_consumer >
void fib_tuner::depends( const int & tag, fib_context & c, dependency_consumer & dC )
const
{
    // we have item-dependencies only if tag > 1
    if( tag > 1 ) {
      dC.depends( c.m_fibs, tag - 1 );
      dC.depends( c.m_fibs, tag - 2 );
    }
}
```

You can find the full code here: fib_tuner.cpp

A similar effect can be achieved by pre-scheduling a step, which executes a step on the same thread which puts the prescribing tag until the first unavailable item was accessed. Obviously this mechanism will not exploit parallelism if all items are available, because the entire step will be executed. Still, this mechanism is much simpler and can yield performance improvements.

Pre-scheduling is enabled by providing a tuner which provides the method "pre-schedule" returning true (fib_preschedule.cpp):

```
struct fib_tuner : public CnC::step_tuner<>
{
    bool preschedule() const { return true; }
};
```

In combination with the unsafe version of get and context::flush_gets() pre-scheduling allows interesting things. For example you can inhibit step execution beyond a call to context::flush_gets() within the pre-scheduling phase, which might increase parallelism. Just uncomment

```
// ctxt.flush_gets(); // uncomment this line to prohibit pre-scheduling from completing
full step-execution
```

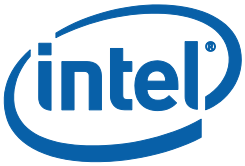
and see how the scheduler statistics change.

TUNING ITEM-COLLECTIONS

Without additional information the runtime cannot decide when no more gets to a data item will be issued. However, Without this information it is not possible to remove a data item from the internal storage. As the CnC programming model never declares "when" something happens, it is also not generally possible to let the user indicate which is the last get of an item.

As keeping data items in memory for ever can quickly (and unnecessarily) blow the available memory a mechanism preventing this is needed. When putting an data item, the CnC C++ API lets you declare the number of gets that will ever be issued. When this "get-count" number of gets has been reached the CnC runtime will remove the item from its internal store and free the memory for other use.

In our Fibonacci example we know that each intermediate item 'x' will be accessed exactly twice: once by fib(x+1) and once again by fib(x+2). Hence the get-count for every Fibonacci item is exactly 2.



The get-count is an attribute of an item, hence the necessary functionality is located in the item-tuners. Declaration of the get-count is straight-forward by providing a const method actually putting the item:

```
int item_tuner::get_count( const int & tag ) const
{
    return tag > 0 ? 2 : 1;
}
```

Like with tag- and step-collections, the tuner needs to made available in the item-collection definition:

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

Each data item will now be freed once it was accessed the second time. The full code (fib_getcount.cpp, fib.h) also accounts for the corner case 0 but also ignores the smaller real get-count for n and n-1.

The get-count features helps a lot keeping the memory-footprint under control and hence can significantly improve application performance.

TAG/STEP MEMOIZATION

As the execution of step-instances is state-less, e.g. functional, executing the same function more than once does not alter the result; it only adds unnecessary overhead. In Fibonacci, most of the tags which prescribe steps are put many times. By default, the CnC runtime will execute the corresponding steps as many times as the same tag is put. By providing a simple hint to the runtime, it will automatically elide duplicate steps. This feature is not enabled by default, because duplicate tags are not common and the automatic memoization apparently adds some overhead.

The memoization is a matter of the tag-collections, hence the corresponding tuning feature is provided by assigning a tuner to the tag-collection. The memoization is enabled by requesting to preserve tags. CnC provides special base-class for tag-tuners to make this extremely convenient:

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

That's all what's needed to let the runtime memoize step executions. In the Fibonacci example this feature has an amazing effect on performance as it elides most of the computation, because due to the nature of the algorithm it stems from redundant computation (also apparent when comparing scheduling statistics).

Next: Tag-Ranges

TAG-RANGES

Often the natural granularity of a step is rather small in the sense of having only low compute load. As the CnC convenience of programming with CnC (e.g. all the automatic parallelism) apparently comes with some overhead, this might not always lead to an efficient program. The CnC C++ API comes with a possibility to prescribe a bunch of tags at once. This feature is closely related to TBB's range concept. Like in TBB, the put range will be subject to recursive splitting/partitioning. Instead of scheduling individual tags, the runtime will then schedule entire ranges and so avoid some overhead induced by the scheduler. Major differences to TBB's range concept are firstly that CnC doesn't really treat it as a consecutive range but only as an iterable set, and secondly that the interface to steps stays at the granularity of individual tags. There is only one interface to steps, e.g. you still define the execute method on tags (and not on tag-ranges) which allows for mixing puts of tags and tag-ranges within the same program.

In the primes example (/samples/primes/primes) we actually put a series of tags to trigger the graph evaluation. Let's use tag-ranges instead; it will give the runtime more freedom for optimization and some might even think it's nicer code. Primarily, tag-ranges as seen as a tuning capability so most of the functionality is within the tuning interface, e.g. in the tag-tuner.

First of all we have to declare the type of range we are going to use. All we need is a blocked range of integers, so let's just use TBB's blocked range. The range-type is declared through a tuner; the tag-tuner interface accepts the range type as its first template argument. As we are not going to provide any custom code, we can just use the CnC tuner-class directly:

```
CnC::tag_collection< int, CnC::tag_tuner< CnC::Internal::strided_range< int > > > m_tags;
```

Now we can put the range instead of writing a for-loop (in main):

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

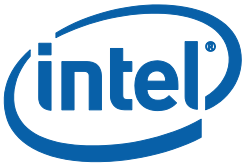
If you run and compile this code (primes_range.cpp) you should observe from the scheduler statistics output that the number of scheduled steps is significantly smaller than the number of checked values. Please note that for this algorithm a more sophisticated partioning strategy would be needed to achieve good scalability. The default partitioning creates equally sized ranges, which will lead to heavier partitions if the tag-values become bigger. Please refer to Advanced Range-Features for more details on the partitioner interface.

CNC::PARALLEL_FOR

The above example checks twice as many values necessary, as it does execute the step also for even numbers. To avoid this, you can define your own range, which could for example use strides or list odd numbers in any other manner (see Advanced Range-Features). Alternatively you can use the convenient parallel_for construct. Besides avoiding unnecessary work, it even simplifies the code: You don't need a tag-collection and instead of prescribing a step you simply apply the step to parallel_for:

```
    CnC::parallel_for( 3, n+1, 2, FindPrimes(), false );
```

The last argument "false" is a tuning parameter (see Tuning Ranges) which tells the runtime that the step-code will not get any items (e.g. is independent of any other steps or items).See primes_parfor.cpp for the full code.



ATTENTION:

In distributed mode, parallel_for will not distribute the given work across processes even if the tuner provides a matching compute_on function. All implied steps-instances will always be scheduled locally.

TUNING RANGES

If you do use ranges or CnC::parallel_for and the step-code does not consume items, you can bypass overhead which is needed to handle data dependencies. The optional template parameter to step_tuner allows this in a very simple manner by setting it to "false":

```
struct my_step_tuner : public CnC::step_tuner< false >
{ ... }
```

A even more simplified version of this parameter is used in CnC::parallel_for and CnC::context::parallel_for, e.g in this example: CnC::parallel_for.

As mentioned in Tag-Ranges the tuner also provides the partitioner. Tag ranges are a matter of tag-collections, so the providing a custom partitioner is done through assigning a tuner to the tag-collection. All you need to do is provide the tuner as the second optional argument to tag_tuner (note that the tuner is needed anyway for ranges):

```
struct my_tag_tuner : public CnC::tag_tuner< my_range_type, my_partitioner_type >
{ ... }
```

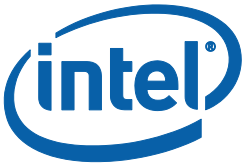
ADVANCED RANGE-FEATURES

Aspects of Tag-Ranges The implemented range-mechanism is very generic. It allows you to provide an arbitrary range as long as you also provide a partitioner which can handle the partitioning of your range (which might be anything, a tree, a matrix etc.). For a detailed description on the requirements on ranges and partitioners please refer to CnC::tag_collection::put_range and CnC::default_partitioner.

It is possible to let a step work on ranges rather than on individual tags. All you need to do is use the range for both, the tag-type and the range-type. To let the runtime use recursive partitioning you also need to provide a partitioner. The API comes with a pre-defined partitioner for this: CnC::tag_partitioner which is otherwise identical to the CnC::default_partitioner. Note: if your tag-type is a range, using CnC::tag_collection::put will not partition the range, only CnC::tag_collection::put_range will do so.

Partitioners are declared through providing a tuner at step-prescription time. Please read CnC::tag_tuner, CnC::default_partitioner and The Tuners for more details.

Next: Runtime Options



RUNTIME OPTIONS

NUMBER OF THREADS

The API allows adjusting the number of worker threads in the code: see CnC::debug::set_num_threads. You can also set the number of threads at runtime by setting the environment variable CNC_NUM_THREADS to the desired number of threads.

NOTE:

The given number must include the environment thread, e.g. n-1 worker threads will be created if CNC_NUM_THREADS is set to n.

Depending on the scheduling strategy, the runtime might actually spawn one or more extra internal helper threads. These threads will mostly be idle and should not have any significant effect on the other threads.

SCHEDULER

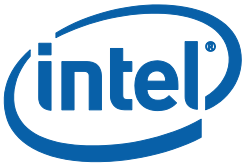
The CnC runtime comes with a selection of different scheduling strategies from which you can chose at runtime. At start-up time, the runtime selects the scheduler upon the evaluation of the environment variable CNC_SCHEDULER. Allowable values are

- TBB_TASK uses the TBB task-stealing scheduler
- TBB_TASK_PRIOR uses the TBB task-stealing scheduler while evaluating tuner::priority
- FIFO_SINGLE uses a single FIFO task-queue
- FIFO_RRIOR_SINGLE uses a single FIFO task-queue while evaluating tuner::priority
- FIFO_STEAL implements task-stealing on thread-local FIFO task-queues
- FIFO_PRIOR_STEAL implements task-stealing on thread-local FIFO task-queues while evaluating tuner::priority

The default scheduler is TBB_TASK. Depending on your application characteristics you might observe significant performance variations with different schedulers. Our experience is that in many cases either TBB_TASK or TBB_FIFO shows best (or best-equivalent) performance.

ATTENTION:

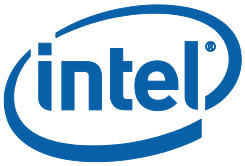
Not all schedulers support nested contexts. When using nested contexts or parallel_for within steps, you might encounter dead-locks when using other schedulers than TBB_TASK[_PRIOR]



BYPASSING THE SCHEDULER

The CnC runtime can optionally execute the first step-instance that is created by the current step (e.g. by putting a tag) right after the currently executing step. This successor step-instance will not go through the normal scheuling procedure and thus avoids overhead. The tradeoff is that it can also limit parallelism. Hence this feature is disabled by default. To enable it set CNC_SCHEDULER_BYPASS to a number which is unequal to 0.

Next: Running CnC applications on distributed memory



RUNNING CNC APPLICATIONS ON DISTRIBUTED MEMORY

In principle, every clean CnC program should be immediately applicable for distributed memory systems. With only a few trivial changes most CnC programs can be made distribution-ready. You will get a binary that runs on shared and distributed memory. Most of the mechanics of data distribution etc. is handled inside the runtime and the programmer does not need to bother about the gory details. Of course, there are a few minor changes needed to make a program distribution-ready, but once that's done, it will run on distributed CnC as well as on "normal" CnC (decided at runtime).

INTER-PROCESS COMMUNICATION

Conceptually, CnC allows data and computation distribution across any kind of network; currently CnC supports SOCKETS and MPI.

LINKING FOR DISTCNC

Support for distributed memory is part of the "normal" CnC distribution, e.g. it comes with the necessary communication libraries (cnc_socket, cnc_mpi). The communication library is loaded on demand at runtime, hence you do not need to link against extra libraries to create distribution-ready applications. Just link your binaries like a "traditional" CnC application (explained in the CnC User Guide, which can be found in the doc directory).

NOTE:

> a distribution-ready CnC application-binary has no dependencies on an MPI library, it can be run on shared memory or over SOCKETS even if no MPI is available on the system

Even though it is not a separate package or module in the CNC kit, in the following we will refer to features that are specific for distributed memory with "distCnC".

MAKING YOUR PROGRAM DISTCNC-READY

As a distributed version of a CnC program needs to do things which are not required in a shared memory version, the extra code for distCnC is hidden from "normal" CnC headers. To include the features required for a distributed version you need to

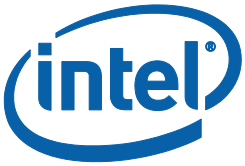
```
#include <cnc/dist_cnc.h>
```

instead of

```
#include <cnc/cnc.h>
```

. If you want to be able to create optimized binaries for shared memory and distributed memory from the same source, you might consider protecting distCnC specifics like this:

```
#ifdef _DIST_
# include <cnc/dist_cnc.h>
#else
# include <cnc/cnc.h>
#endif
```



In "main", initialize an object CnC::dist_cnc_init< list-of-contexts > before anything else; parameters should be all context-types that you would like to be distributed. Context-types not listed in here will stay local. You may mix local and distributed contexts, but in most cases only one context is needed/used anyway.

```
#ifdef _DIST_
    CnC::dist_cnc_init< my_context_type_1 //, my_context_type_2, ...
                      > _dinit;
#endif
```

Even though the communication between process is entirely handled by the CnC runtime, C++ doesn't allow automatic marshalling/serialization of arbitrary data-types. Hence, if and only if your items and/or tags are non-standard data types, the compiler will notify you about the need for serialization/marshalling capability. If you are using standard data types only then marshalling will be handled by CnC automatically.

Marshalling doesn't involve sending messages or alike, it only specifies how an object/variable is packed/unpacked into/from a buffer. Marshalling of structs/classes without pointers or virtual functions can easily be enabled using CNC_BITWISE_SERIALIZABLE(type); others need a "serialize" method or function. The CnC kit comes with an convenient interface for this which is similar to BOOST serialization. It is very simple to use and requires only one function/method for packing and unpacking. See Serialization for more details.

This is it! Your CnC program will now run on distributed memory!

ATTENTION:

Global variables are evil and must not be used within the execution scope of steps. Read Using global read-only data with distCnC about how CnC supports global read-only data. Apparently, pointers are nothing else than global variables and hence need special treatment in distCnC (see Serialization).

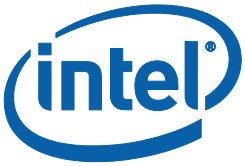
RUNNING DISTCNC

The communication infrastructure used by distCnC is chosen at runtime. By default, the CnC runtime will run your application in shared memory mode. When starting up, the runtime will evaluate the environment variable "DIST_CNC". Currently it accepts the following values

- SHMEM : shared memory (default)
- SOCKETS : communication through TCP sockets
- MPI : using Intel(R) MPI

USING SOCKETS

On application start-up, when DIST_CNC=SOCKETS, CnC checks the environment variable "CNC_SOCKET_HOST". If it is set to a number, it will print a contact string and wait for the given number of clients to connect. Usually



this means that clients need to be started "manually" as follows: set DIST_CNC=SOCKETS and "CNC_SOCKET_CLIENT" to the given contact string and launch the same executable on the desired machine.

If "CNC_SOCKET_HOST" is not a number it is interpreted as a name of a script. CnC executes the script twice: First with "-n" it expects the script to return the number of clients it will start. The second invocation is expected to launch the client processes.

There is a sample script "misc/start.sh" which you can use. Usually all you need is setting the number of clients and replacing "localhost" with the names of the machines you want the application(-clients) to be started on. It requires password-less login via ssh. It also gives some details of the start-up procedure. For windows, the script "start.bat" does the same, except that it will start the clients on the same machine without ssh or alike. Adjust the script to use your preferred remote login mechanism.

MPI

CnC comes with a communication layer based on MPI. You need the Intel(R) MPI runtime to use it. You can download a free version of the MPI runtime from http://software.intel.com/en-us/articles/intel-mpi-library/ (under "Resources"). A distCnC application is launched like any other MPI application with mpirun or mpiexec, but DIST_CNC must be set to MPI:

```
env DIST_CNC=MPI mpiexec -n 4 my_cnc_program
```

Alternatively, just run the app as usually (with DIST_CNC=MPI) and control the number (n) of additionally spawned processes with CNC_MPI_SPAWN=n. If host and client applications need to be different, set CNC_MPI_EXECUTABLE to the client-program name. Here's an example:

```
env DIST_CNC=MPI env CNC_MPI_SPAWN=3 env CNC_MPI_EXECUTABLE=cnc_client cnc_host
```

It starts your host executable "cnc_host" and then spawns 3 additional processes which all execute the client executable "cnc_client".

Please see Using Intel(R) Trace Analyzer and Collector with CnC on how to profile distributed programs.

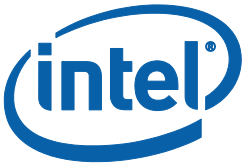
DEFAULT DISTRIBUTION

Step instances are distributed across clients and the host. By default, they are distributed in a round-robin fashion. Note that every process can put tags (and so prescribe new step instances). The round-robin distribution decision is made locally on each process (not globally).

If the same tag is put multiple times, the default scheduling might execute the multiply prescribed steps on different processes and the preserveTags attribute of tag_collections will then not have the desired effect.

The CnC tuning interface provides convenient ways to control the distribution of work and data across the address spaces. The tuning interface is separate from the actual step-code and its declarative nature allows flexible and productive experiments with different distribution strategies (Tuning for distributed memory).

TUNING FOR DISTRIBUTED MEMORY



DISTRIBUTING THE WORK

You can specify the work distribution across the network by providing a tuner to step-collections (the second template argument to CnC::step_collection, see The Tuners). Each step-collection is controlled by its own tuner which maximizes flexibility. By deriving a tuner from CnC::step_tuner it has access to information specific for distributed memory. CnC::tuner_base::numProcs() and CnC::tuner_base::myPid() allow a generic and flexible definition of distribution functions.

To define the distribution of step-instances (the work), your tuner must provide a method called "compute_on", which takes the tag of the step and the context as arguments and has to return the process number to run the step on. To avoid the afore-mentioned problem, you simply need to make sure that the return value is independent of the process it is executed on. The compute_on mechanism can be used to provide any distribution strategy which can be computed locally.

```
   struct my_tuner : public CnC::step_tuner<>
   {
       int compute_on( const tag_type & tag, context_type & ) const { return tag %
numProcs(); }
   };
```

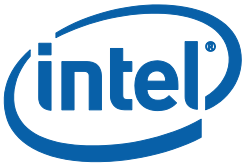
CnC provides special values to make working with compute_on more convenient, more generic and more effective: CnC::COMPUTE_ON_LOCAL, CnC::COMPUTE_ON_ROUND_ROBIN, CnC::COMPUTE_ON_ALL, CnC::COMPUTE_ON_ALL_OTHERS.

DISTRIBUTING THE DATA

By default, the CnC runtime will deliver data items automatically to where they are needed. In its current form, the C++ API does not express the dependencies between instances of steps and/or items. Hence, without additional information, the runtime does not know what step-instances produce and consume which item-instances. Even when the step-distribution is known automatically automatic distribution of data requires global communication. Apparently this constitutes a considerable bottleneck. The CnC tuner interface provides two ways to reduce this overhead.

The ideal, most flexible and most efficient approach is to map items to their consumers. It will convert the default pull-model to a push-model: whenever an item becomes produced, it will be sent only to those processes, which actually need it without any other communication/synchronization. If you can determine which steps are going to consume a given item, you can use the above compute_on to map the consumer step to the actual address spaces. This allows changing the distribution at a single place (compute_on) and the data distribution will be automatically optimized to the minimum needed data transfer.

The runtime evaluates the tuner provided to the item-collection when an item is put. If its method consumed_on (from CnC::item_tuner) returns anything other than CnC::CONSUMER_UNKNOWN it will send the item to the returned process id and avoid all the overhead of requesting the item when consumed.



```
struct my_tuner : public CnC::item_tuner< tag_type, item_type >
{
    int consumed_on( const tag_type & tag )
    {
        return my_step_tuner::consumed_on( consumer_step );
    }
};
```

As more than one process might consume the item, you can also return a vector of ids (instead of a single id) and the runtime will send the item to all given processes.

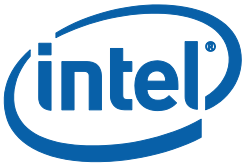
```
struct my_tuner : public CnC::item_tuner< tag_type, item_type >
{
    std::vector< int > consumed_on( const tag_type & tag )
    {
        std::vector< int > consumers;
        foreach( consumer_step of tag ) {
            int _tmp = my_step_tuner::consumed_on( consumer_step );
            consumers.push_back( _tmp );
        }
        return consumers;
    }
};
```

Like for compute_on, CnC provides special values to facilitate and generalize the use of consumed_on: CnC::CONSUMER_UNKNOWN, CnC::CONSUMER_LOCAL, CnC::CONSUMER_ALL and CnC::CONSUMER_ALL_OTHERS.

Note that consumed_on can return CnC::CONSUMER_UNKOWN for some item-instances, and process rank(s) for others.

Sometimes the program semantics make it easier to think about the producer of an item. CnC provides a mechanism to keep the pull-model but allows declaring the owner/producer of the item. If the producer of an item is specified the CnC-runtime can significantly reduce the communication overhead because it on longer requires global communication to find the owner of the item. For this, simply define the depends-method in your step-tuner (derived from CnC::step_tuner) and provide the owning/producing process as an additional argument.

```
struct my_tuner : public CnC::step_tuner<>
{
    int produced_on( const tag_type & tag ) const
    {
        return producer_known ? my_step_tuner::consumed_on( tag ) : tag % numProcs();
    }
};
```



Like for consumed_on, CnC provides special values CnC::PRODUCER_UNKNOWN and CnC::PRODUCER_LOCAL to facilitate and generalize the use of produced_on.

The push-model consumed_on smoothly cooperates with the pull-model as long as they don't conflict.

KEEPING DATA AND WORK DISTRIBUTION IN SYNC

For a more productive development, you might consider implementing consumed_on by thinking about which other steps (not processes) consume the item. With that knowledge you can easily use the appropriate compute_on function to determine the consuming process. The great benefit here is that you can then change compute distribution (e.g. change compute_on) and the data will automatically follow in an optimal way; data and work distribution will always be in sync. It allows experimenting with different distribution plans with much less trouble and lets you define different strategies at a single place. Here is a simple example code which lets you select different strategies at runtime. Adding a new strategy only requires extending the compute_on function: blackscholes.h A more complex example is this one: cholesky.h

USING GLOBAL READ-ONLY DATA WITH DISTCNC

Many algorithms require global data that is initialized once and during computation it stays read-only (dynamic single assignment, DSA). In principle this is aligned with the CnC methodology as long as the initialization is done from the environment. The CnC API allows global DSA data through the context, e.g. you can store global data in the context, initialize it there and then use it in a read-only fashion within your step codes.

The internal mechanism works as follows: on remote processes the user context is default constructed and then de-serialized/un-marshalled. On the host, construction and serialization/marshalling is done in a lazy manner, e.g. not before something actually needs being transferred. This allows creating contexts on the host with non-default constructors, but it requires overloading the serialize method of the context. The actual time of transfer is not statically known, the earliest possible time is the first item- or tag-put. All changes to the context until that point will be duplicated remotely, later changes will not.

Here is a simple example code which uses this feature: blackscholes.h

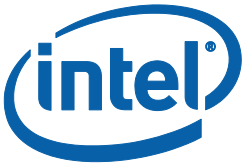
USING INTEL(R) TRACE ANALYZER AND COLLECTOR WITH CNC

A CnC program run can be traced with Intel(R) Trace Collector (ITC) and post-morten analyzed with Intel(R) Trace Analyzer (ITA) (http://software.intel.com/en-us/articles/intel-trace-analyzer/)

The CnC header files are instrumented with hooks into ITC; when compiling with CNC_WITH_ITAC being defined the compiler will insert calls into ITC. Those work in shared and distributed mode, but most (if not all) of the provided instrumentations is for distCnC.

While the runtime provides ITAC versions of the different communication libraries you have to call VT_initialize manually to enable ITC without distCnC. Everything else is handled by the runtime/compiler.

Please see Instrumenting with/for ITAC on how to instrument your code further (e.g. to visualize the actual user code; non-distributed CnC apps).



After instrumentation/recompilation the execution of your applicaton "app" will let ITC write a tracefile "app.stf". To start analysis, simply execute "traceanalyzer app.stf".

HINTS

For a more convinient file handling it is recommended to let ITC write the trac in the "SINGLESTF" format: simply set VT_LOGFILE_FORMAT=SINGLESTF when runnig your application.

Usually CnC codes are threaded. ITC will create a trace in which every thread is shown as busy from it's first acitivity to its termination, even if it is actually idle. To prevent this set VT_ENTER_USCERCODE=0 at application runtime.

INTEL(R) MPI AND ITC

The switch "-trace" to mpi[exec|run|cc] does not work out-of-the-box. However, if using the instrumented version ITC will load automatically. For this, compile with -DCNC_WITH_ITAC and link against ITAC (see Compiling and linking with ITAC instrumentation). To load ITC in a "plain" MPI environment (without recompiling or -linking), just preload the following libraries:

```
env LD_PRELOAD="libcnc.so:libcnc_mpi_itac.so:libVT.so" env DIST_CNC=MPI mpiexec -n ...
```

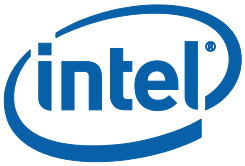
or in debug mode

```
env LD_PRELOAD="libcnc_debug.so:libcnc_mpi_itac_debug.so:libVT.so" env DIST_CNC=MPI
mpiexec -n ...
```

COMPILING AND LINKING WITH ITAC INSTRUMENTATION

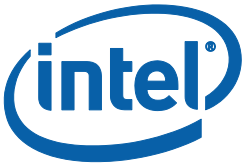
Apparently you need a working isntallation of ITAC. When compiling add the include-directory of ITAC to your include path (-I$VT_ROOT/include). When linking, add the slib directory to the lib-path and link against the following libraries

- SOCKETS:-L$VT_SLIB_DIR -lVTcs $VT_ADD_LIBS
- MPI: use mpicxx/mpiicpc -trace -L$VT_SLIB_DIR Please refer to the ITAC documentation for more details.

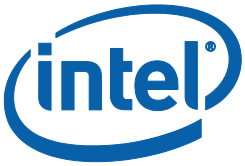


FIBTUTORIAL.CPP

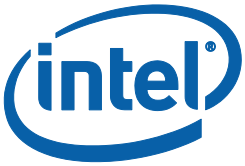
```
//*********************************************************************************
// Copyright (c) 2007-2012 Intel Corporation. All rights reserved.                **
//                                                                                **
// Redistribution and use in source and binary forms, with or without             **
// modification, are permitted provided that the following conditions are met:    **
//   * Redistributions of source code must retain the above copyright notice,     **
//     this list of conditions and the following disclaimer.                      **
//   * Redistributions in binary form must reproduce the above copyright          **
//     notice, this list of conditions and the following disclaimer in the        **
//     documentation and/or other materials provided with the distribution.       **
//   * Neither the name of Intel Corporation nor the names of its contributors    **
//     may be used to endorse or promote products derived from this software      **
//     without specific prior written permission.                                 **
//                                                                                **
// This software is provided by the copyright holders and contributors "as is"    **
// and any express or implied warranties, including, but not limited to, the      **
// implied warranties of merchantability and fitness for a particular purpose     **
// are disclaimed. In no event shall the copyright owner or contributors be       **
// liable for any direct, indirect, incidental, special, exemplary, or            **
// consequential damages (including, but not limited to, procurement of           **
// substitute goods or services; loss of use, data, or profits; or business       **
// interruption) however caused and on any theory of liability, whether in        **
// contract, strict liability, or tort (including negligence or otherwise)        **
// arising in any way out of the use of this software, even if advised of         **
// the possibility of such damage.                                                **
//*********************************************************************************

// compute fibonacci numbers
//

#define _CRT_SECURE_NO_DEPRECATE // to keep the VS compiler happy with TBB

#include <cnc/cnc.h>

// let's use a large type to store fib numbers
typedef unsigned long long fib_type;
// forward declaration
struct fib_context;

// declaration of compute step class
struct fib_step
{
    // declaration of execute method goes here
```



```
    int execute( const int & tag, fib_context & c ) const;
};

// this is our context containing collections and defining their depndencies
struct fib_context : public CnC::context< fib_context > // derive from CnC::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 );
}

// the actual step code computing the fib numbers goes here
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;
```

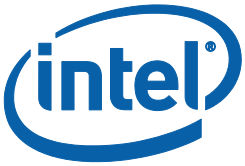


```
}

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;
}
```

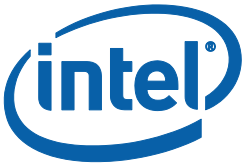


FIB_TRACE.CPP

```
//********************************************************************************
// Copyright (c) 2007-2012 Intel Corporation. All rights reserved.               **
//                                                                               **
// Redistribution and use in source and binary forms, with or without            **
// modification, are permitted provided that the following conditions are met:   **
//   * Redistributions of source code must retain the above copyright notice,    **
//     this list of conditions and the following disclaimer.                     **
//   * Redistributions in binary form must reproduce the above copyright         **
//     notice, this list of conditions and the following disclaimer in the       **
//     documentation and/or other materials provided with the distribution.      **
//   * Neither the name of Intel Corporation nor the names of its contributors   **
//     may be used to endorse or promote products derived from this software     **
//     without specific prior written permission.                                **
//                                                                               **
// This software is provided by the copyright holders and contributors "as is"   **
// and any express or implied warranties, including, but not limited to, the     **
// implied warranties of merchantability and fitness for a particular purpose    **
// are disclaimed. In no event shall the copyright owner or contributors be      **
// liable for any direct, indirect, incidental, special, exemplary, or           **
// consequential damages (including, but not limited to, procurement of          **
// substitute goods or services; loss of use, data, or profits; or business      **
// interruption) however caused and on any theory of liability, whether in       **
// contract, strict liability, or tort (including negligence or otherwise)       **
// arising in any way out of the use of this software, even if advised of        **
// the possibility of such damage.                                               **
//********************************************************************************

// compute fibonacci numbers
//

#define _CRT_SECURE_NO_DEPRECATE // to keep the VS compiler happy with TBB

// let's use a large type to store fib numbers
typedef unsigned long long fib_type;

#include "fib.h"

// the actual step code computing the fib numbers goes here
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;
```

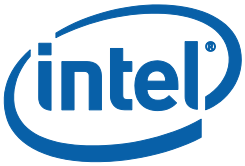


```
        default :
            // get previous 2 results
            fib_type f_1; ctxt.m_fibs.get( tag - 1, f_1 );
            fib_type f_2; ctxt.m_fibs.get( tag - 2, f_2 );
            // put our result
            ctxt.m_fibs.put( tag, f_1 + f_2 );
    }
    return CnC::CNC_Success;
}

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;

        // enable debug output for steps
    CnC::debug::trace( ctxt.m_steps );
    // also enable debug output for our items
        CnC::debug::trace( ctxt.m_fibs );

    // 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;
}
```

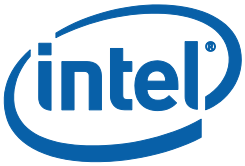


FIB.H

```
//*********************************************************************************
// Copyright (c) 2007-2012 Intel Corporation. All rights reserved.                **
//                                                                                **
// Redistribution and use in source and binary forms, with or without             **
// modification, are permitted provided that the following conditions are met:    **
//   * Redistributions of source code must retain the above copyright notice,     **
//     this list of conditions and the following disclaimer.                      **
//   * Redistributions in binary form must reproduce the above copyright          **
//     notice, this list of conditions and the following disclaimer in the        **
//     documentation and/or other materials provided with the distribution.       **
//   * Neither the name of Intel Corporation nor the names of its contributors    **
//     may be used to endorse or promote products derived from this software      **
//     without specific prior written permission.                                 **
//                                                                                **
// This software is provided by the copyright holders and contributors "as is"    **
// and any express or implied warranties, including, but not limited to, the      **
// implied warranties of merchantability and fitness for a particular purpose     **
// are disclaimed. In no event shall the copyright owner or contributors be       **
// liable for any direct, indirect, incidental, special, exemplary, or            **
// consequential damages (including, but not limited to, procurement of           **
// substitute goods or services; loss of use, data, or profits; or business       **
// interruption) however caused and on any theory of liability, whether in        **
// contract, strict liability, or tort (including negligence or otherwise)        **
// arising in any way out of the use of this software, even if advised of         **
// the possibility of such damage.                                                **
//*********************************************************************************
#ifndef fib_H_ALREADY_INCLUDED
#define fib_H_ALREADY_INCLUDED

#include <cnc/cnc.h>
#include <cnc/debug.h>

// Forward declaration of the context class (also known as graph)
struct fib_context;

// The step classes

struct fib_step
{
    int execute( const int & t, fib_context & c ) const;
};

// The context class
```

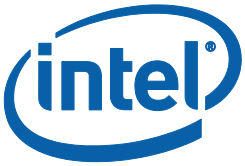


```
struct fib_context : public CnC::context< fib_context >
{
    // the step collection for the instances of the compute-kernel
    CnC::step_collection< fib_step >      m_steps;
    // Item collections
    CnC::item_collection< int, fib_type > m_fibs;
    // Tag collections
    CnC::tag_collection< int >            m_tags;

    // The context class constructor
    fib_context()
        : CnC::context< fib_context >(),
          // Initialize each step collection
          m_steps( *this, "fib_step" ),
          // Initialize each tag collection
          m_fibs( *this, "fibs" ),
          // Initialize each item collection
          m_tags( *this, "tags" )
    {
        // 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 );
    }
};

#endif // fib_H_ALREADY_INCLUDED
```

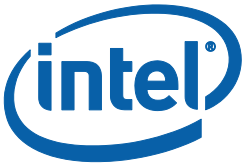


FIB_STATS.CPP

```
//*********************************************************************************
// Copyright (c) 2010-2012 Intel Corporation. All rights reserved.                **
//                                                                                **
// Redistribution and use in source and binary forms, with or without             **
// modification, are permitted provided that the following conditions are met:    **
//   * Redistributions of source code must retain the above copyright notice,     **
//     this list of conditions and the following disclaimer.                      **
//   * Redistributions in binary form must reproduce the above copyright          **
//     notice, this list of conditions and the following disclaimer in the        **
//     documentation and/or other materials provided with the distribution.       **
//   * Neither the name of Intel Corporation nor the names of its contributors    **
//     may be used to endorse or promote products derived from this software      **
//     without specific prior written permission.                                 **
//                                                                                **
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"    **
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE      **
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE     **
// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE       **
// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR            **
// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF           **
// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS       **
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN        **
// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)        **
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF         **
// THE POSSIBILITY OF SUCH DAMAGE.                                                **
//*********************************************************************************

// compute fibonacci numbers
//

#define _CRT_SECURE_NO_DEPRECATE // to keep the VS compiler happy with TBB

// let's use a large type to store fib numbers
typedef unsigned long long fib_type;

#include "fib.h"

// the actual step code computing the fib numbers goes here
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;
```

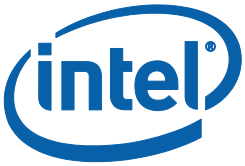


```
        default :
            // get previous 2 results
            fib_type f_1; ctxt.m_fibs.get( tag - 1, f_1 );
            fib_type f_2; ctxt.m_fibs.get( tag - 2, f_2 );
            // put our result
            ctxt.m_fibs.put( tag, f_1 + f_2 );
    }
    return CnC::CNC_Success;
}

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;

    // show scheduler statistics when done
    CnC::debug::collect_scheduler_statistics( 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;
}
```

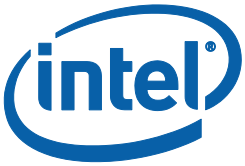


FIB.H

```
//*********************************************************************************
// Copyright (c) 2007-2012 Intel Corporation. All rights reserved.                **
//                                                                                **
// Redistribution and use in source and binary forms, with or without             **
// modification, are permitted provided that the following conditions are met:    **
//   * Redistributions of source code must retain the above copyright notice,     **
//     this list of conditions and the following disclaimer.                      **
//   * Redistributions in binary form must reproduce the above copyright          **
//     notice, this list of conditions and the following disclaimer in the        **
//     documentation and/or other materials provided with the distribution.       **
//   * Neither the name of Intel Corporation nor the names of its contributors    **
//     may be used to endorse or promote products derived from this software      **
//     without specific prior written permission.                                 **
//                                                                                **
// This software is provided by the copyright holders and contributors "as is"    **
// and any express or implied warranties, including, but not limited to, the      **
// implied warranties of merchantability and fitness for a particular purpose     **
// are disclaimed. In no event shall the copyright owner or contributors be       **
// liable for any direct, indirect, incidental, special, exemplary, or            **
// consequential damages (including, but not limited to, procurement of           **
// substitute goods or services; loss of use, data, or profits; or business       **
// interruption) however caused and on any theory of liability, whether in        **
// contract, strict liability, or tort (including negligence or otherwise)        **
// arising in any way out of the use of this software, even if advised of         **
// the possibility of such damage.                                                **
//*********************************************************************************
#ifndef fib_H_ALREADY_INCLUDED
#define fib_H_ALREADY_INCLUDED

#include <cnc/cnc.h>
#include <cnc/debug.h>

// Forward declaration of the context class (also known as graph)
struct fib_context;

// The step classes

struct fib_step
{
    int execute( const int & t, fib_context & c ) const;
};

// The context class
```

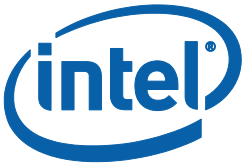


```
struct fib_context : public CnC::context< fib_context >
{
    // the step collection for the instances of the compute-kernel
    CnC::step_collection< fib_step >      m_steps;
    // Item collections
    CnC::item_collection< int, fib_type > m_fibs;
    // Tag collections
    CnC::tag_collection< int >            m_tags;

    // The context class constructor
    fib_context()
        : CnC::context< fib_context >(),
          // Initialize each step collection
          m_steps( *this, "fib_step" ),
          // Initialize each tag collection
          m_fibs( *this, "fibs" ),
          // Initialize each item collection
          m_tags( *this, "tags" )
    {
        // 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 );
    }
};

#endif // fib_H_ALREADY_INCLUDED
```

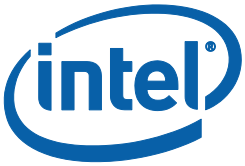


FIB_TUNER.CPP

```
//*********************************************************************************
// Copyright (c) 2010-2012 Intel Corporation. All rights reserved.                **
//                                                                                **
// Redistribution and use in source and binary forms, with or without             **
// modification, are permitted provided that the following conditions are met:    **
//   * Redistributions of source code must retain the above copyright notice,     **
//     this list of conditions and the following disclaimer.                      **
//   * Redistributions in binary form must reproduce the above copyright          **
//     notice, this list of conditions and the following disclaimer in the        **
//     documentation and/or other materials provided with the distribution.       **
//   * Neither the name of Intel Corporation nor the names of its contributors    **
//     may be used to endorse or promote products derived from this software      **
//     without specific prior written permission.                                 **
//                                                                                **
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"    **
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE      **
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE     **
// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE       **
// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR            **
// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF           **
// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS       **
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN        **
// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)        **
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF         **
// THE POSSIBILITY OF SUCH DAMAGE.                                                **
//*********************************************************************************

// compute fibonacci numbers
//

#define _CRT_SECURE_NO_DEPRECATE // to keep the VS compiler happy with TBB

// let's use a large type to store fib numbers
typedef unsigned long long fib_type;

#include "cnc/cnc.h"

struct fib_context;

// let's use a tuner to pre-declare dependencies
struct fib_tuner : public CnC::step_tuner<>
{
    template< class dependency_consumer >
```

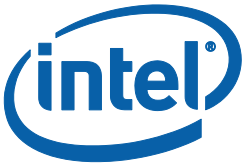


```
    void depends( const int & tag, fib_context & c, dependency_consumer & dC ) const;
};

#include "fib.h"

template< class dependency_consumer >
void fib_tuner::depends( const int & tag, fib_context & c, dependency_consumer & dC )
const
{
    // we have item-dependencies only if tag > 1
    if( tag > 1 ) {
      dC.depends( c.m_fibs, tag - 1 );
      dC.depends( c.m_fibs, tag - 2 );
    }
}

// the actual step code computing the fib numbers goes here
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;
}

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;

    // show scheduler statistics when done
    CnC::debug::collect_scheduler_statistics( ctxt );
```

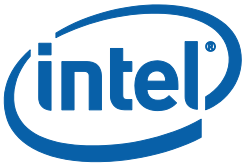


```
    // put tags to initiate evaluation
    for( int i = 0; i <= n; ++i ) ctxt.m_tags.put( i );

    // 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;
}
```

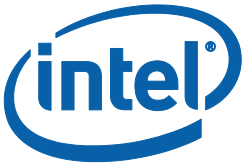
FIB_PRESCHEDULE.CPP



```
// arising in any way out of the use of this software, even if advised of       **
// the possibility of such damage.                                              **
//******************************************************************************

// compute fibonacci numbers
//

#define _CRT_SECURE_NO_DEPRECATE // to keep the VS compiler happy with TBB

// let's use a large type to store fib numbers
typedef unsigned long long fib_type;

#include "cnc/cnc.h"

struct fib_context;

// let's use a tuner to pre-declare dependencies
struct fib_tuner : public CnC::step_tuner<>
{
    bool preschedule() const { return true; }
};

#include "fib.h"

// the actual step code computing the fib numbers goes here
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 );
            // ctxt.flush_gets(); // uncomment this line to prohibit pre-scheduling from
completing full step-execution
            // put our result
            ctxt.m_fibs.put( tag, f_1 + f_2 );
    }
    return CnC::CNC_Success;
}

int main( int argc, char* argv[] )
{
    int n = 42;
    // eval command line args
```

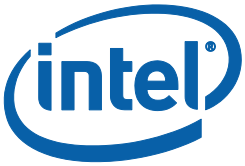


```
    if( argc < 2 ) {
        std::cerr << "usage: " << argv[0] << " n\nUsing default value " << n << std::endl;
    } else n = atol( argv[1] );

    // create context
    fib_context ctxt;

    // show scheduler statistics when done
    CnC::debug::collect_scheduler_statistics( 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;
}
```

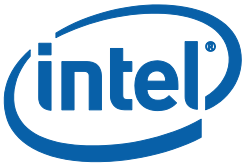


FIB_GETCOUNT.CPP

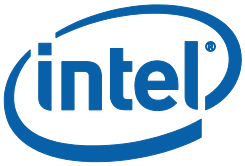
```
//********************************************************************************
// Copyright (c) 2010-2012 Intel Corporation. All rights reserved.              **
//                                                                              **
// Redistribution and use in source and binary forms, with or without           **
// modification, are permitted provided that the following conditions are met:  **
//   * Redistributions of source code must retain the above copyright notice,   **
//     this list of conditions and the following disclaimer.                    **
//   * Redistributions in binary form must reproduce the above copyright        **
//     notice, this list of conditions and the following disclaimer in the      **
//     documentation and/or other materials provided with the distribution.     **
//   * Neither the name of Intel Corporation nor the names of its contributors  **
//     may be used to endorse or promote products derived from this software    **
//     without specific prior written permission.                               **
//                                                                              **
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"  **
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE    **
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE   **
// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE     **
// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR          **
// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF         **
// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS     **
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN      **
// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)      **
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF       **
// THE POSSIBILITY OF SUCH DAMAGE.                                              **
//********************************************************************************

// compute fibonacci numbers
//

#define _CRT_SECURE_NO_DEPRECATE // to keep the VS compiler happy with TBB

#include <cnc/cnc.h>

// let's use a large type to store fib numbers
typedef unsigned long long fib_type;

struct fib_context;

// let's use a tuner to pre-declare dependencies
struct fib_tuner : public CnC::step_tuner<>
{
    // pre-declare data-dependencies
```



```
    template< class dependency_consumer >
    void depends( const int & tag, fib_context & c, dependency_consumer & dC ) const;
};

struct item_tuner : public CnC::hashmap_tuner
{
    // provide number gets to each item
    int get_count( const int & tag ) const;
};

#include "fib.h"

template< class dependency_consumer >
void fib_tuner::depends( const int & tag, fib_context & c, dependency_consumer & dC )
const
{
    // we have item-dependencies only if tag > 1
    if( tag > 1 ) {
      dC.depends( c.m_fibs, tag - 1 );
      dC.depends( c.m_fibs, tag - 2 );
    }
}

int item_tuner::get_count( const int & tag ) const
{
    return tag > 0 ? 2 : 1;
}

// the actual step code computing the fib numbers goes here
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, and specify that this data item will be
                        // get'ed twice (i.e. for the next two fib calculations)
                        // before it is destroyed
            ctxt.m_fibs.put( tag, f_1 + f_2 );
    }
    return CnC::CNC_Success;
}
```

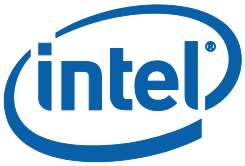


```
int main( int argc, char* argv[] )
{
    int n = 42;
    // eval command line args
    if( argc < 2 ) {
        std::cerr << "usage: " << argv[0] << " n\nUsing default value " << n << std::endl;
    } else n = atol( argv[1] );

    // create context
    fib_context ctxt;

    // show scheduler statistics when done
    CnC::debug::collect scheduler statistics( 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;
}
```

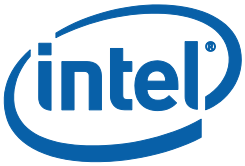


FIB.H

```
//*********************************************************************************
// Copyright (c) 2007-2012 Intel Corporation. All rights reserved.                **
//                                                                                **
// Redistribution and use in source and binary forms, with or without             **
// modification, are permitted provided that the following conditions are met:    **
//   * Redistributions of source code must retain the above copyright notice,     **
//     this list of conditions and the following disclaimer.                      **
//   * Redistributions in binary form must reproduce the above copyright          **
//     notice, this list of conditions and the following disclaimer in the        **
//     documentation and/or other materials provided with the distribution.       **
//   * Neither the name of Intel Corporation nor the names of its contributors    **
//     may be used to endorse or promote products derived from this software      **
//     without specific prior written permission.                                 **
//                                                                                **
// This software is provided by the copyright holders and contributors "as is"    **
// and any express or implied warranties, including, but not limited to, the      **
// implied warranties of merchantability and fitness for a particular purpose     **
// are disclaimed. In no event shall the copyright owner or contributors be       **
// liable for any direct, indirect, incidental, special, exemplary, or            **
// consequential damages (including, but not limited to, procurement of           **
// substitute goods or services; loss of use, data, or profits; or business       **
// interruption) however caused and on any theory of liability, whether in        **
// contract, strict liability, or tort (including negligence or otherwise)        **
// arising in any way out of the use of this software, even if advised of         **
// the possibility of such damage.                                                **
//*********************************************************************************
#ifndef fib_H_ALREADY_INCLUDED
#define fib_H_ALREADY_INCLUDED

#include <cnc/cnc.h>
#include <cnc/debug.h>

// Forward declaration of the context class (also known as graph)
struct fib_context;

// The step classes

struct fib_step
{
    int execute( const int & t, fib_context & c ) const;
};

// The context class
```

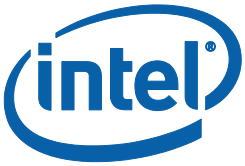


```
struct fib_context : public CnC::context< fib_context >
{
    // the step collection for the instances of the compute-kernel
    CnC::step_collection< fib_step, fib_tuner > m_steps;
    // Item collections
    CnC::item_collection< int, fib_type, item_tuner > m_fibs;
    // Tag collections
    CnC::tag_collection< int >            m_tags;

    // The context class constructor
    fib_context()
        : CnC::context< fib_context >(),
          // Initialize each step collection
          m_steps( *this, "fib_step" ),
          // Initialize each tag collection
          m_fibs( *this, "fibs" ),
          // Initialize each item collection
          m_tags( *this, "tags" )
    {
        // 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 );
    }
};

#endif // fib_H_ALREADY_INCLUDED
```

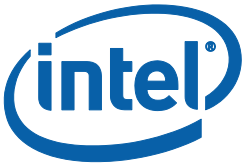


PRIMES_RANGE.CPP

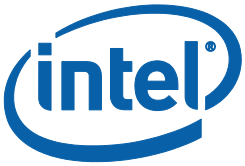
```
//*********************************************************************************
// Copyright (c) 2007-2012 Intel Corporation. All rights reserved.                **
//                                                                                **
// Redistribution and use in source and binary forms, with or without             **
// modification, are permitted provided that the following conditions are met:    **
//   * Redistributions of source code must retain the above copyright notice,     **
//     this list of conditions and the following disclaimer.                      **
//   * Redistributions in binary form must reproduce the above copyright          **
//     notice, this list of conditions and the following disclaimer in the        **
//     documentation and/or other materials provided with the distribution.       **
//   * Neither the name of Intel Corporation nor the names of its contributors    **
//     may be used to endorse or promote products derived from this software      **
//     without specific prior written permission.                                 **
//                                                                                **
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"    **
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE      **
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE     **
// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE       **
// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR            **
// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF           **
// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS       **
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN        **
// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)        **
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF         **
// THE POSSIBILITY OF SUCH DAMAGE.                                                **
//*********************************************************************************
//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <tbb/tick_count.h>
#ifdef _DIST_
#include <cnc/dist_cnc.h>
#include <cnc/internal/dist/distributor.h>
#else
#include <cnc/cnc.h>
#endif


struct my_context;

struct FindPrimes
{
```



```
    int execute( int n, my_context & c ) const;
};

struct primes_tuner : public CnC::step_tuner<>
{
    int compute_on( const int & p, my_context & ) const
    {
        return ( 1 + p / ( 1000000 / 20 ) ) % numProcs();//( p * 5 / 3 + 1 ) % 5;
    }
};

struct my_context : public CnC::context< my_context >
{
    CnC::step_collection< FindPrimes, primes_tuner > m_steps;
    CnC::tag_collection< int, CnC::tag_tuner< CnC::Internal::strided_range< int > > >
m_tags;
    CnC::item_collection< int,int > m_primes;
    my_context()
        : CnC::context< my_context >(),
          m_steps( *this ),
          m_tags( *this ),
          m_primes( *this )
    {
        m_tags.prescribes( m_steps, *this );
    }
};


int FindPrimes::execute( int n, my_context & c ) const
{
    int factor = 3;

    while ( n % factor ) factor += 2;
    if (factor == n) c.m_primes.put(n, n);

    return CnC::CNC_Success;
}


int main(int argc, char* argv[])
{
#ifdef _DIST_
    CnC::dist_cnc_init< my_context > dc_init;
#endif
    bool verbose = false;
    int n = 0;
```

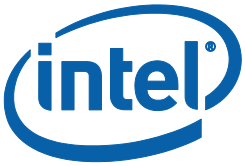


```
    int number_of_primes = 0;

    if (argc == 2)
    {
        n = atoi(argv[1]);
    }
    else if (argc == 3 && 0 == strcmp("-v", argv[1]))
    {
        n = atoi(argv[2]);
        verbose = true;
    }
    else
    {
        fprintf(stderr,"Usage: primes [-v] n\n");
        return -1;
    }

    my_context c;

    printf("Determining primes from 1-%d \n",n);

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

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

    c.wait();

    tbb::tick_count t1 = tbb::tick_count::now();
    // FIXME we have to transfer the items to the host first (distCnC)
    number_of_primes = (int)c.m_primes.size() + 1;
    printf("Found %d primes in %g seconds\n", number_of_primes, (t1-t0).seconds());

    if (verbose)
    {
        printf("%d\n", 2);
        CnC::item_collection<int,int>::const_iterator cii;
        for (cii = c.m_primes.begin(); cii != c.m_primes.end(); cii++)
        {
            printf("%d\n", cii->first); // kludge
        }
    }
}
```

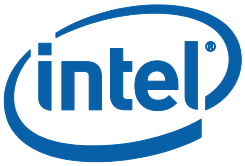


PRIMES_PARFOR.CPP

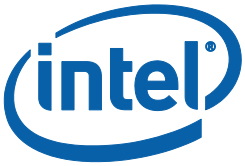
```
//*********************************************************************************
// Copyright (c) 2007-2012 Intel Corporation. All rights reserved.                **
//                                                                                **
// Redistribution and use in source and binary forms, with or without             **
// modification, are permitted provided that the following conditions are met:    **
//   * Redistributions of source code must retain the above copyright notice,     **
//     this list of conditions and the following disclaimer.                      **
//   * Redistributions in binary form must reproduce the above copyright          **
//     notice, this list of conditions and the following disclaimer in the        **
//     documentation and/or other materials provided with the distribution.       **
//   * Neither the name of Intel Corporation nor the names of its contributors    **
//     may be used to endorse or promote products derived from this software      **
//     without specific prior written permission.                                 **
//                                                                                **
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"    **
// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE      **
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE     **
// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE       **
// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR            **
// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF           **
// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS       **
// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN        **
// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)        **
// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF         **
// THE POSSIBILITY OF SUCH DAMAGE.                                                **
//*********************************************************************************
//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <tbb/tick_count.h>
#include <cnc/cnc.h>
#include <cnc/debug.h>

struct my_context;

struct FindPrimes
{
    int operator()( int n ) const;
};
```



```
struct my_context : public CnC::context< my_context >
{
    CnC::item_collection< int,int > m_primes;
    my_context()
        : CnC::context< my_context >(),
          m_primes( *this )
    {
        CnC::debug::collect_scheduler_statistics( *this );
    }
};


my_context g_c;

int FindPrimes::operator()( int n ) const
{
    int factor = 3;

    while ( n % factor ) factor += 2;
    if (factor == n) g_c.m_primes.put(n, n);

    return CnC::CNC_Success;
}


int main(int argc, char* argv[])
{
    bool verbose = false;
    int n = 0;
    int number_of_primes = 0;

    if (argc == 2)
    {
        n = atoi(argv[1]);
    }
    else if (argc == 3 && 0 == strcmp("-v", argv[1]))
    {
        n = atoi(argv[2]);
        verbose = true;
    }
    else
    {
        fprintf(stderr,"Usage: primes [-v] n\n");
        return -1;
    }
```

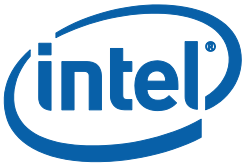


```
    printf("Determining primes from 1-%d \n",n);

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

    CnC::parallel_for( 3, n+1, 2, FindPrimes(), false );

    tbb::tick_count t1 = tbb::tick_count::now();

    number_of_primes = (int)g_c.m_primes.size() + 1;
    printf("Found %d primes in %g seconds\n", number_of_primes, (t1-t0).seconds());

    if (verbose)
    {
        printf("%d\n", 2);
        for (CnC::item_collection<int,int>::const_iterator cii =
g_c.m_primes.begin(); cii != g_c.m_primes.end(); cii++)
        {
            printf("%d\n", cii->first); // kludge
        }
    }
}
```

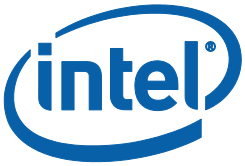


BLACKSCHOLES.H

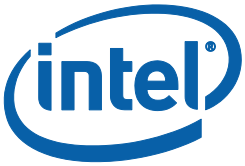
```
//*********************************************************************************
// Copyright (c) 2007-2012 Intel Corporation. All rights reserved.                **
//                                                                                **
// Redistribution and use in source and binary forms, with or without             **
// modification, are permitted provided that the following conditions are met:    **
//   * Redistributions of source code must retain the above copyright notice,     **
//     this list of conditions and the following disclaimer.                      **
//   * Redistributions in binary form must reproduce the above copyright          **
//     notice, this list of conditions and the following disclaimer in the        **
//     documentation and/or other materials provided with the distribution.       **
//   * Neither the name of Intel Corporation nor the names of its contributors    **
//     may be used to endorse or promote products derived from this software      **
//     without specific prior written permission.                                 **
//                                                                                **
// This software is provided by the copyright holders and contributors "as is"    **
// and any express or implied warranties, including, but not limited to, the      **
// implied warranties of merchantability and fitness for a particular purpose     **
// are disclaimed. In no event shall the copyright owner or contributors be       **
// liable for any direct, indirect, incidental, special, exemplary, or            **
// consequential damages (including, but not limited to, procurement of           **
// substitute goods or services; loss of use, data, or profits; or business       **
// interruption) however caused and on any theory of liability, whether in        **
// contract, strict liability, or tort (including negligence or otherwise)        **
// arising in any way out of the use of this software, even if advised of         **
// the possibility of such damage.                                                **
//*********************************************************************************
#ifndef blackscholes_H_ALREADY_INCLUDED
#define blackscholes_H_ALREADY_INCLUDED

#ifdef _DIST_
# include <cnc/dist_cnc.h>
#else
# include <cnc/cnc.h>
#endif
#include <cnc/debug.h>
#include <vector>
#include <memory>

typedef std::shared_ptr< std::vector< OptionData > > option_vector_type;
typedef std::shared_ptr< std::vector< fptype > > price_vector_type;

CNC_BITWISE_SERIALIZABLE( OptionData );
```



```
// Forward declaration of the context class (also known as graph)
struct blackscholes_context;

// The step classes
struct Compute
{
    int execute( const int &, blackscholes_context & ) const;
};

struct bs_tuner : public CnC::step_tuner<>, public CnC::vector_tuner
{
    int compute_on( const int tag ) const {
        return tag % numProcs();
    }
    int compute_on( const int tag, blackscholes_context & ) const {
        return compute_on( tag );
    }
    int consumed_by( const int tag ) const {
        return tag;
    }
    int consumed_on( const int tag ) const {
        return compute_on( consumed_by( tag ) );
    }
    int getcount( const int tag ) {
        return 1;
    }
};

// The context class
struct blackscholes_context : public CnC::context< blackscholes_context >
{
    // Step collections
    CnC::step_collection< Compute, bs_tuner > compute;

    // Item collections
    CnC::item_collection< int, option_vector_type, bs_tuner > opt_data;
    CnC::item_collection< int, price_vector_type, bs_tuner >  prices;

    // Tag collections
    CnC::tag_collection< int > tags;

    int m_vs;
    // The context class constructor
    blackscholes_context( int vs = 0)
        : CnC::context< blackscholes_context >(),
          // Initialize each step collection
```

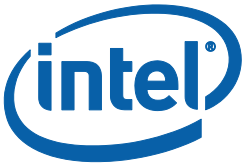


```
          compute( *this ),
          // Initialize each item collection
          opt_data( *this ),
          prices( *this ),
          // Initialize each tag collection
          tags( *this ),
          m_vs( vs )
    {
        // Prescriptive relations
        tags.prescribes( compute, *this );
        compute.produces( prices );
        opt_data.set_max( m_vs );
        prices.set_max( m_vs );
    }
#ifdef _DIST_
    void serialize( CnC::serializer & ser )
    {
        ser & m_vs;
        if( ser.is_unpacking() ) {
            opt_data.set_max( m_vs );
            prices.set_max( m_vs );
        }

    }
#endif
};

#endif // blackscholes_H_ALREADY_INCLUDED
```

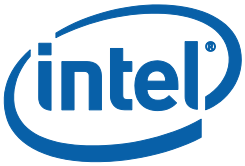


CHOLESKY.H

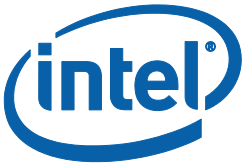
```
//********************************************************************************
// Copyright (c) 2007-2012 Intel Corporation. All rights reserved.               **
//                                                                               **
// Redistribution and use in source and binary forms, with or without            **
// modification, are permitted provided that the following conditions are met:   **
//   * Redistributions of source code must retain the above copyright notice,    **
//     this list of conditions and the following disclaimer.                     **
//   * Redistributions in binary form must reproduce the above copyright         **
//     notice, this list of conditions and the following disclaimer in the       **
//     documentation and/or other materials provided with the distribution.      **
//   * Neither the name of Intel Corporation nor the names of its contributors   **
//     may be used to endorse or promote products derived from this software     **
//     without specific prior written permission.                                **
//                                                                               **
// This software is provided by the copyright holders and contributors "as is"   **
// and any express or implied warranties, including, but not limited to, the     **
// implied warranties of merchantability and fitness for a particular purpose    **
// are disclaimed. In no event shall the copyright owner or contributors be      **
// liable for any direct, indirect, incidental, special, exemplary, or           **
// consequential damages (including, but not limited to, procurement of          **
// substitute goods or services; loss of use, data, or profits; or business      **
// interruption) however caused and on any theory of liability, whether in       **
// contract, strict liability, or tort (including negligence or otherwise)       **
// arising in any way out of the use of this software, even if advised of        **
// the possibility of such damage.                                               **
//********************************************************************************
#ifndef cholesky_H_ALREADY_INCLUDED
#define cholesky_H_ALREADY_INCLUDED

#ifdef _DIST_
# include <cnc/dist_cnc.h>
#else
# include <cnc/cnc.h>
#endif
#include <cnc/debug.h>
#include <memory>

// Forward declaration of the context class (also known as graph)
struct cholesky_context;

// The step classes

struct S1_compute
```



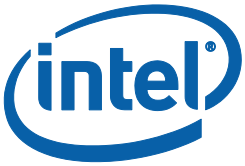
```
{
    int execute( const int & t, cholesky_context & c ) const;
};

struct S2_compute
{
    int execute( const pair & t, cholesky_context & c ) const;
};

struct S3_compute
{
    int execute( const triple & t, cholesky_context & c ) const;
};

struct k_compute
{
    int execute( const int & t, cholesky_context & c ) const;
};

struct kj_compute
{
    int execute( const int & t, cholesky_context & c ) const;
};

struct kji_compute
{
    int execute( const pair & t, cholesky_context & c ) const;
};

static void mark( int p, bool * d )
{
    if( d[p] == false ) {
        d[p] = true;
    }
}

struct cholesky_tuner : public CnC::step_tuner<>, public CnC::hashmap_tuner
{
    cholesky_tuner( cholesky_context & c, int p = 0, int n = 0, dist_type dt = BLOCKED_ROWS
)
        : m_context( c ), m_p( p ), m_n( n ),
          m_div( (((p*p) / 2) + 1) / numProcs() ),
          //          m_div( p ? (((p*(p+1))/2)+((2*numProcs())/2))/(2*numProcs()) :
0 ),
          m_dt( dt )
    {
```



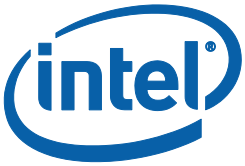
```
        if( myPid() == 0 ) {
            switch( dt ) {
            default:
            case BLOCKED_ROWS :
                std::cerr << "Distributing BLOCKED_ROWS\n";
                break;
            case ROW_CYCLIC :
                std::cerr << "Distributing ROW_CYCLIC\n";
                break;
            case COLUMN_CYCLIC :
                std::cerr << "Distributing COLUMN_CYCLIC\n";
                break;
            case BLOCKED_CYCLIC :
                std::cerr << "Distributing BLOCKED_CYCLICS\n";
                break;
            }
        }
    }

    inline static int compute_on( const dist_type dt, const int i, const int j, const
int n, const int s )
    {
        switch( dt ) {
        default:
        case BLOCKED_ROWS :
            return ( ((j*j)/2 + 1 + i ) / s ) % numProcs();
            break;
        case ROW_CYCLIC :
            return j % numProcs();
            break;
        case COLUMN_CYCLIC :
            return i % numProcs();
            break;
        case BLOCKED_CYCLIC :
            return ( (i/2) * n + (j/2) ) % numProcs();
            break;
        }
    }

    // step-bits
    int compute_on( const int tag, cholesky_context & /*arg*/ ) const
    {
        return compute_on( m_dt, tag, tag, m_n, m_div );
    }

    int compute_on( const pair & tag, cholesky_context & /*arg*/ ) const
```



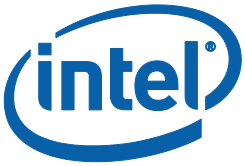
```
    {
        return compute_on( m_dt, tag.first, tag.second, m_p, m_div );
    }

    int compute_on( const triple & tag, cholesky_context & /*arg*/ ) const
    {
        return compute_on( m_dt, tag[2], tag[1], m_p, m_div );
    }

    // item-bits
    typedef triple tag_type;

    int get_count( const tag_type & tag ) const
    {
        int _k = tag[0], _i = tag[2];
        if( _k == _i+1 ) return CnC::NO_GETCOUNT; // that's our result
        return ( _k > 0 && _k > _i ) ? ( m_p - _k ) : 1;
    }

    // First we determine which *steps* are going to consume this item.
    // Then we use compute_on to determine the *processes* to which the item needs to
go.
    // Mostly the two steps are formulated in one line.
    // We avoid duplicate entries by using a helper mechanism "mark"
    std::vector< int > consumed_on( const tag_type & tag  ) const
    {
        int _k = tag[0], _j = tag[1], _i = tag[2];

        if( _i == _j ) { // on diagonal
            if( _i == _k ) return std::vector< int >( 1, compute_on( _k, m_context )
); // S1 only
            if( _k == m_p ) return std::vector< int >( 1, 0 ); // the end
        }
        if( _i == _k ) return std::vector< int >( 1, compute_on( pair( _k, _j ), m_context
) ); // S2 only

        bool * _d;
        _d = new bool[numProcs()];
        memset( _d, 0, numProcs() * sizeof( *_d ) );

        if( _i == _k-1 ) {
            if( _i == _j ) { // on diagonal on S2
                for( int j = _k; j < m_p; ++j ) {
                    mark( compute_on( pair( _k - 1, j ), m_context ), _d );
                }
            } else { // S3
```



```
                for( int j = _j; j < m_p; ++j ) {
                    for( int i = _k; i <= _j; ++i ) {
                        mark( compute_on( triple( _k-1, j, i ), m_context ), _d );
                    }
                }
            }
        }
        mark( compute_on( triple( _k, _j, _i ), m_context ), _d );
        std::vector< int > _v;
        _v.reserve( numProcs()/2 );
        if( _d[myPid()] ) _v.push_back( myPid() );
        for( int i = 0; i < numProcs(); ++i ) {
            if( _d[i] && i != myPid() ) _v.push_back( i );
        }
        delete [] _d;
        return _v;
    }

#ifdef _DIST_
    void serialize( CnC::serializer & ser )
    {
        ser & m_p & m_div & m_n & m_dt;
    }
#endif
private:
    cholesky_context & m_context;
    int m_p;
    int m_n;
    int m_div;
    dist_type m_dt;
};

// The context class
struct cholesky_context : public CnC::context< cholesky_context >
{
    // tuners
    cholesky_tuner tuner;

    // Step Collections
    CnC::step_collection< S1_compute, cholesky_tuner > sc_s1_compute;
    CnC::step_collection< S2_compute, cholesky_tuner > sc_s2_compute;
    CnC::step_collection< S3_compute, cholesky_tuner > sc_s3_compute;
    CnC::step_collection< k_compute, cholesky_tuner > sc_k_compute;
    CnC::step_collection< kj_compute, cholesky_tuner > sc_kj_compute;
    CnC::step_collection< kji_compute, cholesky_tuner > sc_kji_compute;
```



```
    // Item collections
    CnC::item_collection< triple, tile_const_ptr_type, cholesky_tuner > Lkji;
    int p,b;

    // Tag collections
    CnC::tag_collection< int > control_S1;
    CnC::tag_collection< pair > control_S2;
    CnC::tag_collection< triple > control_S3;
    CnC::tag_collection< int > singleton;

    // The context class constructor
    cholesky_context( int _b = 0, int _p = 0, int _n = 0, dist_type dt = BLOCKED_ROWS
)
        : CnC::context< cholesky_context >(),
          // init tuners
          tuner( *this, _p, _n, dt ),
          // init step colls
          sc_s1_compute( *this, tuner, "Cholesky" ),
          sc_s2_compute( *this, tuner, "Trisolve" ),
          sc_s3_compute( *this, tuner, "Update" ),
          sc_k_compute( *this, tuner ),
          sc_kj_compute( *this, tuner ),
          sc_kji_compute( *this, tuner ),
          // Initialize each item collection
          Lkji( *this, "Lkji", tuner ),
          p( _p ),
          b( _b ),
          // Initialize each tag collection
          control_S1( *this, "S1" ),
          control_S2( *this, "S2" ),
          control_S3( *this, "S3" ),
          singleton( *this )
    {
        // Prescriptive relations
        singleton.prescribes( sc_k_compute, *this );
        control_S1.prescribes( sc_s1_compute, *this );
        control_S1.prescribes( sc_kj_compute, *this );
        control_S2.prescribes( sc_s2_compute, *this );
        control_S2.prescribes( sc_kji_compute, *this );
        control_S3.prescribes( sc_s3_compute, *this );

        // control relations
        sc_k_compute.controls( control_S1 );
        sc_kj_compute.controls( control_S2 );
        sc_kji_compute.controls( control_S3 );
```



```
        // producer/consumer relations
        sc_s1_compute.consumes( Lkji );
        sc_s1_compute.produces( Lkji );
        sc_s2_compute.consumes( Lkji );
        sc_s2_compute.produces( Lkji );
        sc_s3_compute.consumes( Lkji );
        sc_s3_compute.produces( Lkji );
#if 0
        CnC::debug::trace all( *this );
#endif
    }

#ifdef _DIST_
    void serialize( CnC::serializer & ser )
    {
        ser & p & b & tuner;
    }
#endif
};

#endif // cholesky_H_ALREADY_INCLUDED
```

MODULE INDEX

MODULES

Here is a list of all modules:

NAMESPACE INDEX

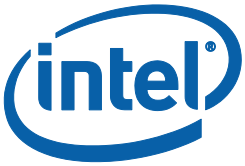
NAMESPACE LIST

Here is a list of all documented namespaces with brief descriptions:

CLASS INDEX

CLASS HIERARCHY

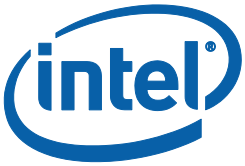
This inheritance list is sorted roughly, but not completely, alphabetically:



CLASS INDEX

CLASS LIST

Here are the classes, structs, unions and interfaces with brief descriptions:



MODULE DOCUMENTATION

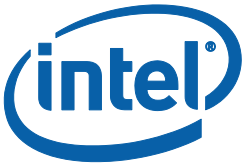
SERIALIZATION

MODULES

- Serialization of simple structs/classes without pointers or virtual functions.
- Non bitwise serializable objects
- Marshalling pointer types (e.g. items which are pointers)
- Automatic serialization of built-in types.

CLASSES

- class serializable
- class explicitly_serializable



- *Specifies serialization category: explicit serialization via a "serialize" function.* class bitwise_serializable
- *simple structs/classes are bitwise serializable.* class chunk< T, Allocator >
- *Serialization of arrays with and without automatic memory handling.* class serializer

HANDLES SERILIALIZATION OF DATA-OBJECTS. MACROS

- #define CNC_BITWISE_SERIALIZABLE(T)
- #define CNC_POINTER_SERIALIZABLE(T)

FUNCTIONS

- template<class T > explicitly_serializable serializer_category (const T *)

DETAILED DESCRIPTION

There are three possible ways of setting up the serialization of your struct resp. class:

MACRO DEFINITION DOCUMENTATION

#DEFINE CNC_BITWISE_SERIALIZABLE(T)

Value:
```
inline CnC::bitwise_serializable serializer_category( const T * ) { \
            return CnC::bitwise_serializable(); \
        }
```

Convenience macro for defining that a type should be treated as bitwise serializable:

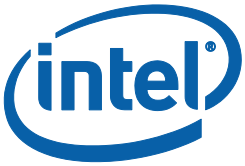
Definition at line 226 of file serializer.h.

#DEFINE CNC_POINTER_SERIALIZABLE(T)

Value:
```
namespace CnC {                                                 \
        void serialize( serializer & ser, T *& t ) {  \
            ser & chunk< T >(t, 1);                          \
        }                                                                 \
    }
```

Convenience macro for declaring serializable pointers. The object being pointed to should be serializable.

Definition at line 288 of file serializer.h.



FUNCTION DOCUMENTATION

EXPLICITLY_SERIALIZABLE CNC::SERIALIZER_CATEGORY (CONST T *) [INLINE]

General case: a type belongs to the category explicitly_serializable, unless there is a specialization of the function template "serializer_category" (see below).

Definition at line 213 of file serializer.h.

```
    {
        return explicitly_serializable();
    }
```

SERIALIZATION OF SIMPLE STRUCTS/CLASSES WITHOUT POINTERS OR VIRTUAL FUNCTIONS.

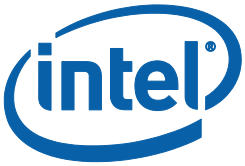
```
   If your struct does not contain pointers (nor virtual
   functions), instances of it may be serialized by simply
   copying them bytewise. In this case, it suffices to
   add your struct to the category bitwise_serializable which is done
   by providing the following global function:
   @code
inline bitwise_serializable serializer_category( const Your_Class * ) {
    return bitwise_serializable();
}
```

As a short-cut, you can just use CNC_BITWISE_SERIALIZABLE(T) to declare your class as bitwise serializable:

```
    CNC_BITWISE_SERIALIZABLE( my_class );
```

NON BITWISE SERIALIZABLE OBJECTS

```
   You'll have to provide a serializer function for your class.
   One way of doing this is to derive your class from
   serializable and to provide a method "serialize":
   @code
class Your_class : public serializable {
    ...
public:
    void serialize( CnC::serializer & );
    ...
};
```



In the implementation of this serialize method you should use the operator& of the serializer class. Here is an example:

```
class Your_class : public serializable {
private:
    double x;
    float  y;
    int    len;
    char*  arr; // array of lenth len
    ...
public:
    void serialize( CnC::serializer & buf ) {
        buf & x
            & y
            & len;
        buf & CnC::chunk< char >( arr, len_arr ); // chunk declared below
        ...
    }
}
```

This method will called for both packing and unpacking (what will actually be done, depends on the internal mode of the serializer.)

Alternatively, you can provide a global function

```
void serialize( serializer& buf, Your_class& obj );
```

which should use the operator& of the serializer class (same as for method 2. above). This is useful in cases where you want to leave your class unchanged.

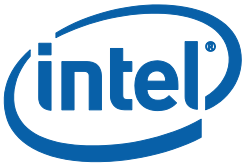
MARSHALLING POINTER TYPES (E.G. ITEMS WHICH ARE POINTERS)

MODULES

- Serialization of std::shared_ptr

DETAILED DESCRIPTION

```
A common optimization used by CnC programs targeting the
shared-memory runtime is to store pointers to large objects
inside of item collections instead of copies of the objects.
This is done to minimize the overhead of copying into and out
```



```
of the item collection.  Programs written in this style can be
sued with distCnC, but special care needs to be taken.
\n
Distributed CnC requires that data stored in item collections be
serializable in order to communicate data between different
processes.  Consequently, the programmer must modify such
pointer-using CnC programs in two ways in order to use the
distributed-memory runtime.  First, the programmer must provide a
specialization of CnC::serialize for the pointer type pointing to T:
@code
```

void CnC::serialize(CnC::serializer & ser, T *& ptr) { ser & CnC::chunk< Your_class[, allocator] >(ptr, 1); } Please refer to CnC::chunk to learn about appropriate memory management.

If the pointer represents a single object and not a C-style array, then the programmer may the use convenience macro:

```
CNC_POINTER_SERIALIZABLE( T );
```

This macro implements the above method for the case when the pointer refers to only a single object (length == 1). Note that if T is a type that contains template parameters, the programmer should create a typedef of the specific type being serialized and use this type as the argument for the macro.

Next, programmers must ensure that type T itself implements a default constructor and is serializable by itself. The latter as achieved as described above. It is the same method that programmers must provide if storing copies of objects instead of pointers to objects.

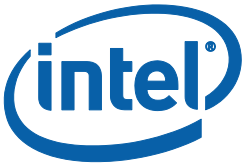
Reading from and writing to item collections is not different in this style of writing CnC programs. However, there are a few things to note:

When an object is communicated from one process to another, an item must be allocated on the receiving end. By default, CnC::chunk uses the default allocator for this and so requires the default constructor of T. After object creation, any fields are updated by the provided serialize methods. Thios default behavior simplifies the object allocation/deallocation: most of it is done automatically behind the scenes. The programmer should just pass an unitialized pointer to get(). The pointer will then point to the allocated object/array after the get completes.

In advanced and potentially dangerous setups, the programmer might wish to store the object into memory that has been previously allocated. This can be done by an explicit copy and explicit object lifetime control. Please note that such a scenario can easily lead to uses of CnC which do not comply with CnC's methodology, e.g. no side effects in steps and dynamic single assigments.

SERIALIZATION OF STD::SHARED_PTR

std::shared_ptr are normally supported out-of-the-box (given the underlying type is serializable (similar to Marshalling pointer types (e.g. items which are pointers)).



Only shared_ptrs to more complex pointer-types require special treatment to guarantee correct garbage collection. If

- using std::shared:ptr
- AND the pointer-class holds data which is dynamically allocated
- AND this dynamic memory is NOT allocated in the default constructor but during marshalling with CnC::chunk

then (and only then) you might want to use construct_array and destruct_array. It is probably most convenient to

- use the standard CnC::chunk mechanism during serialization
- allocate dynamic memory in the constructors with construct_array
- and deallocate it in the destructor using destruct_array

This mechanism is used in the cholesky example which comes with this distribution.

Alternatively you can use matching allocation/deallocation explicitly in your con/destructors and during serialization.

AUTOMATIC SERIALIZATION OF BUILT-IN TYPES.

FUNCTIONS

- CNC_BITWISE_SERIALIZABLE (bool)

  bool is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (char)

  char is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (signed char)

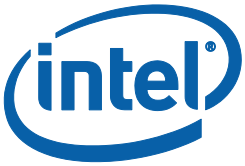
  signed char is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (unsigned char)

  unsigend char is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (short)

  short is bitwise serializable



- CNC_BITWISE_SERIALIZABLE (int)

  int is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (long)

  long is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (unsigned long long)

  unsigned long long is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (float)

  float is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (double)

  double is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (long double)

DETAILED DESCRIPTION

They can be copied byte-wise.

In order to add your own class to the category bitwise_serializable, add a specialization of "serializer_category" returning bitwise_serializable for your class. (e.g. by using CnC::CNC_BITWISE_SERIALIZABLE).

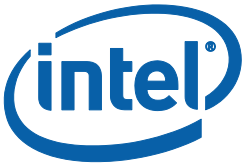
FUNCTION DOCUMENTATION

CNC_BITWISE_SERIALIZABLE (UNSIGNED *CHAR*)

unsigend char is bitwise serializable
unsigned long is bitwise serializable
unsigned int is bitwise serializable
unsigend short is bitwise serializable

CNC_BITWISE_SERIALIZABLE (LONG)

long is bitwise serializable



long long is bitwise serializable

CNC::CNC_BITWISE_SERIALIZABLE (LONG *DOUBLE*)

long double is bitwise serializable

INSTRUMENTING WITH/FOR ITAC

MACROS

- #define VT_THREAD_NAME(threadName)
- #define VT_FUNC(funcName)

DETAILED DESCRIPTION

Two Macros are provided for a convenient way to instrument your code with ITAC. When adding instrumentation you need to take care about building your bainry properly (Compiling and linking with ITAC instrumentation).

MACRO DEFINITION DOCUMENTATION

#DEFINE VT_FUNC(FUNCNAME)

Use this macro to manually instrument your function (or scope) at its beginning, e.g. VT_FUNC("MyClass::myMethod"); See also Compiling and linking with ITAC instrumentation

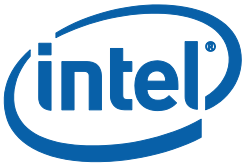
PARAMETERS:

funcName	name of function (in double quotes), potentially including class/namespace hierachy

Definition at line 133 of file itac.h.

#DEFINE VT_THREAD_NAME(THREADNAME)

Use this macro to define an (optional) name for your application thread, e.g. VT_THREAD_NAME("myApp"). This name will appear for your thread in ITAC's event timeline.



PARAMETERS:

threadName	name of calling thread (in double quotes), to appear in ITAC tracefile

Definition at line 127 of file itac.h.

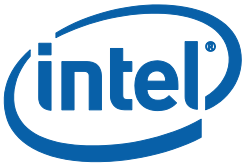
NAMESPACE DOCUMENTATION

CNC NAMESPACE REFERENCE

CnC API.

CLASSES

- class step_collection
- *A step collection is logical set of step instances.* class tag_collection
- *A tag collection is a set of tags of the same type. It is used to prescribe steps. By default, tags are not stored.* class item_collection
- *An item collection is a mapping from tags to items.* class context
- *CnC context bringing together collections (for steps, items and tags).* struct debug
- *Debugging interface providing tracing and timing capabilities.* class tuner_base
- struct step_tuner
- *Default (NOP) implementations of the tuner interface.* struct item_tuner
- *Default implementations of the item-tuner interface for item-collections.* struct hashmap_tuner
- *The tuner base for hashmap-based item-tuners.* struct vector_tuner
- *The tuner base for vector-based item-tuners.* struct tag_tuner
- struct preserve_tuner
- struct dist_cnc_init
- class serializable
- class explicitly_serializable



- *Specifies serialization category: explicit serialization via a "serialize" function.* class bitwise_serializable
- *simple structs/classes are bitwise serializable.* class chunk
- *Serialization of arrays with and without automatic memory handling.* class serializer
- *Handles serilialization of data-objects.* class default_partitioner
- *Interface for partitioners: configuring how ranges are partitioned.* class tag_partitioner

ENUMERATIONS

- enum { COMPUTE_ON_LOCAL = -2, COMPUTE_ON_ROUND_ROBIN = -3, COMPUTE_ON_ALL = -4, COMPUTE_ON_ALL_OTHERS = -5, PRODUCER_UNKNOWN = -6, PRODUCER_LOCAL = -7, CONSUMER_UNKNOWN = -8, CONSUMER_LOCAL = -9, CONSUMER_ALL = -10, CONSUMER_ALL_OTHERS = -11, NO_GETCOUNT = Internal::item_properties::NO_GET_COUNT }

FUNCTIONS

- template<class Index , class Functor > Functor parallel_for (Index first, Index last, Index incr, const Functor &f, bool gets_items=true)

  *Execute f(i) for every i in {first <= i=first+step*x < last and 0 <= x}.*

- template<class T > explicitly_serializable serializer_category (const T *)
- CNC_BITWISE_SERIALIZABLE (bool)

  bool is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (char)

  char is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (signed char)

  signed char is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (unsigned char)

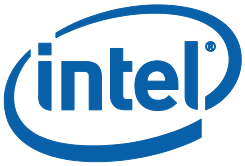
  unsigend char is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (short)

  short is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (int)

  int is bitwise serializable



- CNC_BITWISE_SERIALIZABLE (long)

  long is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (unsigned long long)

  unsigned long long is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (float)

  float is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (double)

  double is bitwise serializable

- CNC_BITWISE_SERIALIZABLE (long double)

VARIABLES

- const int CNC_Success = 0

  Steps return CNC_Success if execution was successful.

- const int CNC_Failure = 1

  Steps return CNC_Failure if execution failed.

DETAILED DESCRIPTION

CnC API.

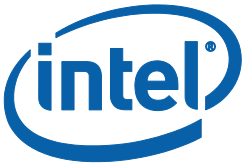
ENUMERATION TYPE DOCUMENTATION

ANONYMOUS ENUM

Enumerator:

COMPUTE_ON_LOCAL let tuner::compute_on return COMPUTE_ON_NO if the step should be executed locally

COMPUTE_ON_ROUND_ROBIN let tuner::compute_on return COMPUTE_ON_ROUND_ROBIN to let the scheduler distribute it in a round-robin fashion



COMPUTE_ON_ALL let tuner::compute_on return COMPUTE_ON_ALL if the step should be executed on all processes, as well as locally

COMPUTE_ON_ALL_OTHERS let tuner::compute_on return COMPUTE_ON_ALL_OTHERS if the step should be executed on all processes, but not locally

PRODUCER_UNKNOWN producer process of dependent item is unknown

PRODUCER_LOCAL producer process of dependent item is local process

CONSUMER_UNKNOWN consumer process of given item is unkown

CONSUMER_LOCAL consumer process of given item is the local process

CONSUMER_ALL all processes consume given item

CONSUMER_ALL_OTHERS all processes but this consume given item

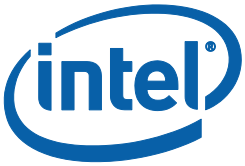
NO_GETCOUNT no get-count specified

Definition at line 47 of file default_tuner.h.

```
          {
        COMPUTE_ON_LOCAL       = -2,  ///< let tuner::compute_on return
COMPUTE_ON_NO if the step should be executed locally
        COMPUTE_ON_ROUND_ROBIN = -3,  ///< let tuner::compute_on return
COMPUTE_ON_ROUND_ROBIN to let the scheduler distribute it in a round-robin fashion
        COMPUTE_ON_ALL         = -4,  ///< let tuner::compute_on return
COMPUTE_ON_ALL if the step should be executed on all processes, as well as locally
        COMPUTE_ON_ALL_OTHERS  = -5,  ///< let tuner::compute_on return
COMPUTE_ON_ALL_OTHERS if the step should be executed on all processes, but not locally
        PRODUCER_UNKNOWN     = -6,  ///< producer process of dependent item is
unknown
        PRODUCER_LOCAL       = -7,  ///< producer process of dependent item is local
process
        CONSUMER_UNKNOWN     = -8,  ///< consumer process of given item is unkown
        CONSUMER_LOCAL       = -9,  ///< consumer process of given item is the local
process
        CONSUMER_ALL         = -10, ///< all processes consume given item
        CONSUMER_ALL_OTHERS = -11, ///< all processes but this consume given item
        NO_GETCOUNT          = Internal::item_properties::NO_GET_COUNT   ///< no
get-count specified
    };
```

FUNCTION DOCUMENTATION

FUNCTOR CNC::PARALLEL_FOR (INDEX *FIRST*, INDEX *LAST*, INDEX *INCR*, CONST FUNCTOR & *F*, BOOL *GETS_ITEMS* = TRUE)



Execute f(i) for every i in {first <= i=first+step*x < last and 0 <= x}.

For different values of i, function execution might occur in parallel. Returns functor object ocne all iterations have been executed. Type Index must support operator+. Executes on the local process only. No distribution to other processes supported.

PARAMETERS:

first	starting index of parallel iteration
last	iteration stops before reaching last
incr	increment index by this value in each iteration
f	function to be executed
gets_items	true (default) if function gets items, false otherwise (might perform better)

CLASS DOCUMENTATION

BITWISE_SERIALIZABLE CLASS REFERENCE

simple structs/classes are bitwise serializable.

DETAILED DESCRIPTION

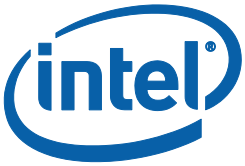
simple structs/classes are bitwise serializable.

Specifies serialization category: byte-wise copy

Definition at line 221 of file serializer.h.

The documentation for this class was generated from the following file:

- serializer.h



CHUNK< T, ALLOCATOR > CLASS TEMPLATE REFERENCE

Serialization of arrays with and without automatic memory handling.

DETAILED DESCRIPTION

TEMPLATE<CLASS T, CLASS ALLOCATOR = STD::ALLOCATOR< T >>CLASS CNC::CHUNK< T, ALLOCATOR >

Serialization of arrays with and without automatic memory handling.

```
   Specified (to serializer::operator&) via
   @code
CnC::chunk< type[, allocator] >( your_ptr, len )
```

where your_ptr is a pointer variable to your array of <type>s.

If the allocator is the special type CnC::no_alloc, the runtime assumes that the programer appropriately allocates (if ser.is_unpacking()) and deallocates (if ser:is_cleaning_up()) the array. This implies that the using serialize method/function needs to make sure it passes valid and correct pointers to chunk.

If not CnC::no_alloc , the Allocator class must meet "allocator" requirements of ISO C++ Standard, Section 20.1.5

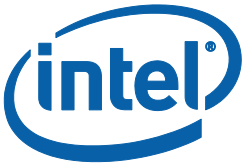
When unpacking, your_ptr must be NULL, and it is allocated automatically using the given allocator and when cleaning up, your_ptr is deallocated accordingly.

If using an allocator (in particular hre default std::allocator) object/array allocation and deallocation is greatly simplified for the programmer. There is no need to allocate and/or deallocate objects/arrays manually.

Definition at line 269 of file serializer.h.

The documentation for this class was generated from the following file:

- serializer.h



CNC_TAG_HASH_COMPARE< T > STRUCT TEMPLATE REFERENCE

Provides hash and equality operators for hashing.

PUBLIC MEMBER FUNCTIONS

- size_t hash (const T &x) const
- bool equal (const T &a, const T &b) const

DETAILED DESCRIPTION

TEMPLATE<CLASS T>STRUCT CNC_TAG_HASH_COMPARE< T >

Provides hash and equality operators for hashing.

Specializations for custom data types must implement hash and equal to work with preserving tag-collections (CnC::preserve_tuner) and/or (default) CnC::item_collection using hashmaps.

Standard data types are supported out-of-the-box as well as std::vector and std:pair thereof, char*, std::string and pointers (which you better avoid if ever possible, because it pointers as tags are not portable, e.g. to distributed memory).

Definition at line 42 of file cnc_tag_hash_compare.h.

MEMBER FUNCTION DOCUMENTATION

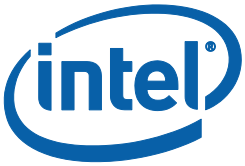
BOOL EQUAL (CONST T & *A*, CONST T & *B*) CONST [INLINE]

RETURNS:

true if a equals b, false otherwise

Definition at line 51 of file cnc_tag_hash_compare.h.

```
    {
        return ( a == b );
    }
```



SIZE_T HASH (CONST T & *X*) CONST [INLINE]

RETURNS:

a unique integer for the given tag

Definition at line 45 of file cnc_tag_hash_compare.h.

```
{
    // Knuth's multiplicative hash
    return static_cast< size_t >( x ) * 2654435761;
}
```

THE DOCUMENTATION FOR THIS STRUCT WAS GENERATED FROM THE FOLLOWING FILE:

- cnc_tag_hash_compare.h

CNC_TAG_HASH_COMPARE< STD::STRING > STRUCT TEMPLATE REFERENCE

hash/equality for std:.string

DETAILED DESCRIPTION

TEMPLATE<>STRUCT CNC_TAG_HASH_COMPARE< STD::STRING >

hash/equality for std:.string

Definition at line 98 of file cnc_tag_hash_compare.h.

The documentation for this struct was generated from the following file:

- cnc_tag_hash_compare.h

SERIALIZER::CONSTRUCT_ARRAY< T, ALLOCATOR > STRUCT TEMPLATE REFERENCE

Allocates an array of type T and size num in pointer variable arrVar.

DETAILED DESCRIPTION

TEMPLATE<CLASS T, CLASS ALLOCATOR = STD::ALLOCATOR< T >>STRUCT CNC::SERIALIZER::CONSTRUCT_ARRAY< T, ALLOCATOR >

Allocates an array of type T and size num in pointer variable arrVar.

Definition at line 420 of file serializer.h.

The documentation for this struct was generated from the following file:

- serializer.h

CONTEXT< DERIVED > CLASS TEMPLATE REFERENCE

CnC context bringing together collections (for steps, items and tags).

PUBLIC MEMBER FUNCTIONS

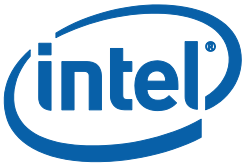
- context ()

  default constructor

- virtual ~context ()

  destructor

- error_type wait ()



wait until all the steps prescribed by this context have completed execution.

- void flush_gets ()

  used with the preschedule tuner to finalize 'gets' in the pre-execution of a step

- void reset ()

  reset all collections of this context

- virtual void serialize (serializer &)

  (distCnC) overload this if default construction on remote processes is not enough.

DETAILED DESCRIPTION

TEMPLATE<CLASS DERIVED>CLASS CNC::CONTEXT< DERIVED >

CnC context bringing together collections (for steps, items and tags).

```
   The user needs to derive his or her own context from the CnC::context.
   The template argument to context is the user's context class itself.

   For example,
   @code
```

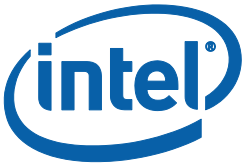
struct my_context : public CnC::context< my_context > { CnC::step_collection< FindPrimes > steps; CnC::tag_collection< int > oddNums; CnC::item_collection< int,int > primes; my_context() : CnC::context< my_context >(), steps(this), oddNums(this), primes(this) { oddNums.prescribes(steps); } };

```
   Several contexts can be created and executed simultaneously.

   Execution starts as soon as a step(-instance) is prescribed through putting a tag.
   All ready steps will be executed even if CnC::context::wait() is never be called.

   It is recommended to declare collections as members of a context derived from
CnC::context.
```

This yields more maintanable code and future versions of CnC may require this convention.

Definition at line 333 of file cnc.h.

MEMBER FUNCTION DOCUMENTATION



VOID FLUSH_GETS ()

used with the preschedule tuner to finalize 'gets' in the pre-execution of a step

Call this after last call to the non-blocking item_collection::unsafe_get method.

ERROR_TYPE WAIT ()

wait until all the steps prescribed by this context have completed execution.

RETURNS:

0 if succeeded, error code otherwise

THE DOCUMENTATION FOR THIS CLASS WAS GENERATED FROM THE FOLLOWING FILE:

- cnc.h

DEBUG STRUCT REFERENCE

Debugging interface providing tracing and timing capabilities.

STATIC PUBLIC MEMBER FUNCTIONS

- static void CNC_API set_num_threads (int n)

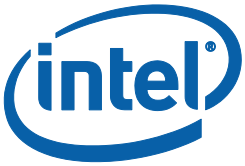
  sets the number of threads used by the application

- template<typename Tag , typename Tuner > static void trace (tag_collection< Tag, Tuner > &tc, int level=1)

  enable tracing of a given tag collection at a given level

- template<typename Tag , typename Item , typename HC > static void trace (item_collection< Tag, Item, HC > &ic, int level=1)

  enable tracing of a given item collection at a given level



- template<typename UserStep , typename Tuner > static void trace (step_collection< UserStep, Tuner > &sc, int level=1)

  enable tracing of a given step-collection at a given level (off=0)

- template<class Derived > static void trace_all (::CnC::context< Derived > &c, int level=1)

  enable tracing of everything in given context (off=0)

- static void init_timer (bool cycle=false)

  initalize timer

- static void finalize_timer (const char *name)

  save collected time log to a specified file

- template<typename UserStep , typename Tuner > static void time (step_collection< UserStep, Tuner > &sc)

  enable timing of a given step

- template<class Derived > static void collect_scheduler_statistics (::CnC::context< Derived > &c)

  enable collection scheduler statistics per context

DETAILED DESCRIPTION

Debugging interface providing tracing and timing capabilities.

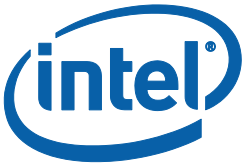
```
\#include <cnc/debug.h>

For a meaningful tracing the runtime requires a function cnc_format for
every non-standard item and tag type. The expected signature is:
@code
```

std::ostream & cnc_format(std::ostream & os, const your_type & p) Make sure it is defined in the corresponding namespace.

Definition at line 46 of file debug.h.

MEMBER FUNCTION DOCUMENTATION

STATIC VOID COLLECT_SCHEDULER_STATISTICS (::CNC::CONTEXT< DERIVED > & *C*) [INLINE], [STATIC]



enable collection scheduler statistics per context

Statistics will be print upon destructino of a context.

PARAMETERS:

c	the context to be examined

Definition at line 118 of file debug.h.

```
{ c.init_scheduler_statistics(); }
```

STATIC VOID FINALIZE_TIMER (CONST CHAR * *NAME*) [INLINE], [STATIC]

save collected time log to a specified file

PARAMETERS:

name	the file to write the time log, pass "-" for printing to stdout

Definition at line 102 of file debug.h.

```
{ Internal::chronometer::save_log( name ? name : "-" ); }
```

STATIC VOID INIT_TIMER (BOOL *CYCLE* = FALSE) [INLINE], [STATIC]

initalize timer

PARAMETERS:

cycle	if true, use cycle counter only Cycle counters might overflow: TSC results are incorrect if the measured time-interval is larger than a full turn-around.

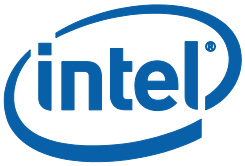
Definition at line 97 of file debug.h.

```
{ Internal::chronometer::init( cycle ); }
```

STATIC VOID CNC_API SET_NUM_THREADS (INT *N*) [STATIC]

sets the number of threads used by the application

Overwrites environment variable CNC_NUM_THREADS. To be effective, it must be called prior to context creation.



STATIC VOID TIME (STEP_COLLECTION< USERSTEP, TUNER > & *SC*) [INLINE], [STATIC]

enable timing of a given step

To be used in a safe environment only (no steps in flight)

PARAMETERS:

sc	the step-collection to be timed

Definition at line 110 of file debug.h.

```
{ sc.set_timing(); }
```

STATIC VOID TRACE (TAG_COLLECTION< TAG, TUNER > & *TC*, INT *LEVEL* = 1) [INLINE], [STATIC]

enable tracing of a given tag collection at a given level

To be used in a safe environment only (no steps in flight)

PARAMETERS:

tc	the tag collection to be traced
level	trace level

Definition at line 59 of file debug.h.

```
{ tc.m_tagCollection.set_tracing( level ); }
```

STATIC VOID TRACE (ITEM_COLLECTION< TAG, ITEM, HC > & *IC*, INT *LEVEL* = 1) [INLINE], [STATIC]

enable tracing of a given item collection at a given level

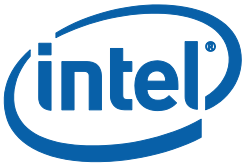
To be used in a safe environment only (no steps in flight).

PARAMETERS:

ic	the item collection to be traced
level	trace level

Definition at line 68 of file debug.h.

```
{ ic.m_itemCollection.set_tracing( level ); }
```



STATIC VOID TRACE (STEP_COLLECTION< USERSTEP, TUNER > & *SC*, INT *LEVEL* = 1)[INLINE], [STATIC]

enable tracing of a given step-collection at a given level (off=0)

To be used in a safe environment only (no steps in flight)

PARAMETERS:

sc	the step-collection to be traced
level	trace level

Definition at line 77 of file debug.h.

```
        {
            sc.set_tracing( level );
            CNC_ASSERT( sc.trace_level() == level );
        }
```

STATIC VOID TRACE_ALL (::CNC::CONTEXT< DERIVED > & *C*, INT *LEVEL* = 1)[INLINE], [STATIC]

enable tracing of everything in given context (off=0)

To be used in a safe environment only (no steps in flight) names of collections are unavailable unless tracing them was enabled explicitly.

PARAMETERS:

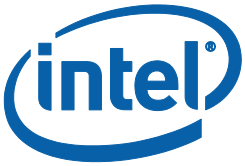
c	the context to be traced
level	trace level

Definition at line 90 of file debug.h.

```
        { c.set_tracing( level ); }
```

THE DOCUMENTATION FOR THIS STRUCT WAS GENERATED FROM THE FOLLOWING FILE:

- debug.h



DEFAULT_PARTITIONER< GRAINSIZE > CLASS TEMPLATE REFERENCE

Interface for partitioners: configuring how ranges are partitioned.

Inherited by tag_partitioner< grainSize >.

PUBLIC TYPES

- typedef range_is_range_type split_type

PUBLIC MEMBER FUNCTIONS

- template<typename Range , typename StepInstance > bool divide_and_originate (Range &range, StepInstance &si) const

  divide given range into in arbitrary number of ranges of type Range

PROTECTED MEMBER FUNCTIONS

- template<typename Range > bool is_divisible (const Range &range) const

  return true, if given range is divisible, false otherwise

- int grain_size (size_t fullRangeSize) const

DETAILED DESCRIPTION

TEMPLATE<INT GRAINSIZE = 0>CLASS CNC::DEFAULT_PARTITIONER< GRAINSIZE >

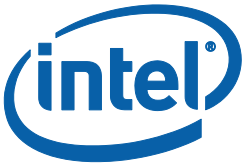
Interface for partitioners: configuring how ranges are partitioned.

The default_partitioner implements the template interface that each partitioner must satisfy.

Given a range type "R", a partitioner "P" must provide a copy-constructor and the following (template) interface:

- template< typename T > void divide_and_originate(R &, T &) const;
- split_type

The default_partitioner can be parametrized as follows: Set template argument to > 0 to let it use a fixed grainsize. If it equals 0, the grainsize is set to "original_rangeSize / #threads / 4" If it is < 0, the grainsize of the given range is obeyed.



Definition at line 47 of file default_partitioner.h.

MEMBER TYPEDEF DOCUMENTATION

TYPEDEF RANGE_IS_RANGE_TYPE SPLIT_TYPE

set to range_is_tag_type if tag is self-dividing, e.g. if the range-type is also the tag-type as passed to the step
set to range_is_range_type if tag is undivisible, e.g. if range-type != step_type

Reimplemented in tag_partitioner< grainSize >.

Definition at line 80 of file default_partitioner.h.

MEMBER FUNCTION DOCUMENTATION

BOOL DIVIDE_AND_ORIGINATE (RANGE & *RANGE*, STEPINSTANCE & *SI*) CONST [INLINE]

divide given range into in arbitrary number of ranges of type Range

Call si.originate_range(r) for each new range. The original - but modified - range must *not* be passed to originate_range! If tag-types are self-dividing (e.g. if range-type == tag-type) you should call "originate" instead of "originate_range" for leaves of the recursive range-tree.

RETURNS:

true if the orignal - but modified - range needs further splitting, false if no further splitting is desired.

The aggregated set of the members of the sub-ranges applied to "t.originate[_range]" must equal the set of member in given range. Overlapping ranges or gaps may lead to arbitrary effects.

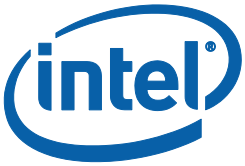
PARAMETERS:

range	the original range to split, may be altered
si	opaque object, call t.originate[_range](r) for all split-off sub-ranges

Reimplemented in tag_partitioner< grainSize >.

Definition at line 139 of file default_partitioner.h.

```
    {
        return this->divide_and_originate( range, si, this->grain_size(
range.size() ) );
```



```
    }
```

INT GRAIN_SIZE (SIZE_T *FULLRANGESIZE*) CONST [INLINE], [PROTECTED]

RETURNS:

grainsize for given size of unsplitted range

Definition at line 108 of file default_partitioner.h.

```
    {
        if( grainSize != 0 ) return grainSize;
        else {
            if( m_grainSize <= 0 ) {
#define MX( _a, _b ) ((_a) > (_b) ? (_a) : (_b))
                const_cast< int & >( m_grainSize ) = rangeSize > 0 ? MX( 1,
(int)(rangeSize / (size_t)m_nt / 4) ) : 1;
            }
            return m_grainSize;
        }
    }
```

THE DOCUMENTATION FOR THIS CLASS WAS GENERATED FROM THE FOLLOWING FILE:

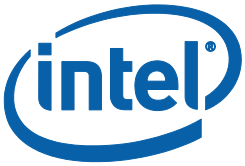
- default_partitioner.h

SERIALIZER::DESTRUCT_ARRAY< T, ALLOCATOR > STRUCT TEMPLATE REFERENCE

destructs the array of type T and isze num at arrVar and resets arrVar to NULL.

DETAILED DESCRIPTION

TEMPLATE<CLASS T, CLASS ALLOCATOR = STD::ALLOCATOR< T >>STRUCT CNC::SERIALIZER::DESTRUCT_ARRAY< T, ALLOCATOR >



destructs the array of type T and isze num at arrVar and resets arrVar to NULL.

Definition at line 426 of file serializer.h.

The documentation for this struct was generated from the following file:

- serializer.h

DIST_CNC_INIT< C1, C2, C3, C4, C5 > STRUCT TEMPLATE REFERENCE

DETAILED DESCRIPTION

TEMPLATE<CLASS C1, CLASS C2 = INTERNAL::VOID_CONTEXT, CLASS C3 = INTERNAL::VOID_CONTEXT, CLASS C4 = INTERNAL::VOID_CONTEXT, CLASS C5 = INTERNAL::VOID_CONTEXT>STRUCT CNC::DIST_CNC_INIT< C1, C2, C3, C4, C5 >

To enable remote CnC you must create one such object right after entering main. The object must persist throughout main. All context classes ever used in the program must be referenced as template arguments. All contexts must have all collections they use as members and must be default-constructible. Pointers as tags are not supported by distCnC.

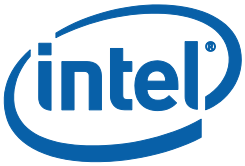
Definition at line 386 of file dist_cnc.h.

The documentation for this struct was generated from the following file:

- dist_cnc.h

EXPLICITLY_SERIALIZABLE CLASS REFERENCE

Specifies serialization category: explicit serialization via a "serialize" function.



DETAILED DESCRIPTION

Specifies serialization category: explicit serialization via a "serialize" function.

Definition at line 207 of file serializer.h.

The documentation for this class was generated from the following file:

- serializer.h

HASHMAP_TUNER STRUCT REFERENCE

The tuner base for hashmap-based item-tuners.

Inherits item_tuner< Internal::hash_item_table >.

ADDITIONAL INHERITED MEMBERS

DETAILED DESCRIPTION

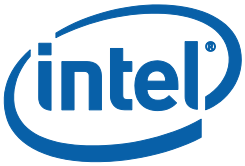
The tuner base for hashmap-based item-tuners.

The internal hash-map uses cnc_tag_hash_compare. If your tag-type is not supported by default, you need to provide a template specialization for it.

Definition at line 241 of file default_tuner.h.

The documentation for this struct was generated from the following file:

- default_tuner.h



ITEM_COLLECTION< TAG, ITEM, TUNER > CLASS TEMPLATE REFERENCE

An item collection is a mapping from tags to items.

PUBLIC TYPES

- typedef Tag tag_ype

  the tag type

PUBLIC MEMBER FUNCTIONS

- template<class Derived > item_collection (context< Derived > &ctxt, const std::string &name, const Tuner &tnr)

  constructor which registers collection with given context

- void set_max (size_t mx)

  Declares the maxium tag value.

- void put (const Tag &tag, const Item &item)

  make copies of the item and the tag and store them in the collection.

- void get (const Tag &tag, Item &item) const

  get an item

- bool unsafe_get (const Tag &tag, Item &item) const

  try to get an item and store it in given object (non-blocking)

- const_iterator begin () const

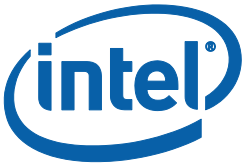
  returns begin() as in STL containers

- const_iterator end () const

  returns end() as in STL containers

- void reset ()

  removes all of the item instances from the collection



- size_t size ()

  returns number of elements in collection

- bool empty ()

  returns true if size()==0, false otherwise

DETAILED DESCRIPTION

TEMPLATE<TYPENAME TAG, TYPENAME ITEM, TYPENAME TUNER = HASHMAP_TUNER>CLASS CNC::ITEM_COLLECTION< TAG, ITEM, TUNER >

An item collection is a mapping from tags to items.

Tag and Item must provide copy and default constructors and the assigment operator.

Th last template argument is an optional tuner. The tuner provides tuning hints, such as the type of the data store or information about its use in distributed environments. Most importantly it tells the runtime and compiler which type of data store it should use. By default that's a hash-map (hashmap_tuner). For non default-supported tag types (e.g. those that are not convertable into size_t) a suitable cnc_tag_hash_compare template specialization must be provided. For the vector-based data store (vector_tuner) that's not necessary, but the tag-type must then be convertible to and from size_t.

SEE ALSO:

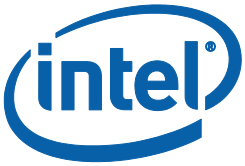
CnC::item_tuner for more information.

The CnC runtime will make a copy of your item when it is 'put' into the item_collection. The CnC runtime will delete the copied item copy once the get-count reaches 0 (or, if no get-count was provided, once the collection is destroyed). If the item-type is a pointer type, the runtime will not delete the memory the item points to. If you store pointeres, you have to care for the appropriate garbage collection, e.g. you might consider using smart pointers.

Definition at line 213 of file cnc.h.

CONSTRUCTOR & DESTRUCTOR DOCUMENTATION

ITEM_COLLECTION (CONTEXT< DERIVED > & *CTXT*, CONST STD::STRING & *NAME*, CONST TUNER & *TNR*)



constructor which registers collection with given context

PARAMETERS:

ctxt	the context this collection belongs to
name	an optional name, used for debug output and tracing
tnr	a tuner object which must persist throughout the lifetime of the step-collection by default a default-constructed tuner will be used.

MEMBER FUNCTION DOCUMENTATION

CONST_ITERATOR BEGIN () CONST

returns begin() as in STL containers

NOTE:

iteration through collections is not thread safe use it only between calls to CnC::context::wait() and putting tags

CONST_ITERATOR END () CONST

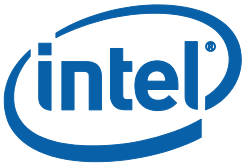
returns end() as in STL containers

NOTE:

iteration through collections is not thread safe use it only between calls to CnC::context::wait() and putting tags

VOID GET (CONST TAG & *TAG*, ITEM & *ITEM*) CONST

get an item



PARAMETERS:

tag	the tag identifying the item
item	reference to item to store result in

EXCEPTIONS:

DataNotReady	throws exception if data not yet available.

VOID PUT (CONST TAG & *TAG*, CONST ITEM & *ITEM*)

make copies of the item and the tag and store them in the collection.

PARAMETERS:

tag	the tag identifying the item
item	the item to be copied and stored

VOID SET_MAX (SIZE_T *MX*)

Declares the maxium tag value.

Must be called prior to accessing the collection if the data store is a vector. Useful only for dense tag-spaces.

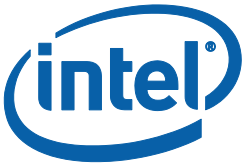
PARAMETERS:

mx	the largest tag-value ever used for this collection

BOOL UNSAFE_GET (CONST TAG & *TAG*, ITEM & *ITEM*) CONST

try to get an item and store it in given object (non-blocking)

ATTENTION:



This method is unsafe: you can create non-deterministic results if you decide to to perform semantically relevant actions if an item is unavailable (returns false)

If the item is unavailable, it does not change item. Make sure you call flush_gets() after last call to this method (of any item collection) within a step. In any case, you must check the return value before accessing the item.

PARAMETERS:

tag	the tag identifying the item
item	reference to item to store result in

RETURNS:

true if item is available

EXCEPTIONS:

DataNotReady	might throw exception if data not available (yet)

THE DOCUMENTATION FOR THIS CLASS WAS GENERATED FROM THE FOLLOWING FILE:

- cnc.h

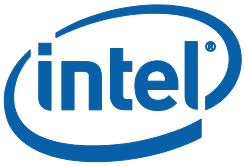
ITEM_TUNER< TT > STRUCT TEMPLATE REFERENCE

Default implementations of the item-tuner interface for item-collections.

Inherits tuner_base.

CLASSES

- struct table_type



DEFINES THE TYPE OF THE INTERNAL DATA STORE. PUBLIC MEMBER FUNCTIONS

- template<typename Tag > int get_count (const Tag &tag) const

  Allows specifying the number of gets to the given item.

- template<typename Tag > int consumed_on (const Tag &tag) const

  Tells the scheduler on which process(es) this item is going to be consumed return process id where the item will be consumed (get), or CONSUMER_UNKNOWN (default) or std::vector<int>, containing all ids of consuming processes. To indicate that the consumer processes are unknown, return an empty vector. The vector must contain special values like CONSUMER_LOCAL. If not CnC::CONSUMER_UKNOWN (or empty vector), this declaration will overwrite what the step-tuner might declare in depends. Providing this method leads to the most efficient communication pattern for to getting the data to were it is needed.

- template<typename Tag > int produced_on (const Tag &tag) const

  Tells the scheduler on which process(es) this item is going to be produced return process id where the item will be produced. If unknown return CnC::PRODUCER_UNKNOWN. return PRODUCER_LOCAL if local process is the owner. Implementing this method reduces the communication cost for locating and sending data. Implementing item_tuner::consumed_on is more efficient, but might be more complicated. Will be evaluated only if item_tuner::consumed_on returns CnC::CONSUMER_UNKNOWN.

ADDITIONAL INHERITED MEMBERS

DETAILED DESCRIPTION

TEMPLATE<TEMPLATE< TYPENAME T, TYPENAME I > CLASS TT>STRUCT CNC::ITEM_TUNER< TT >

Default implementations of the item-tuner interface for item-collections.

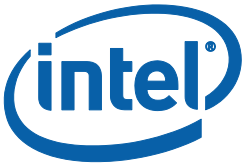
Usually you will not need to use this directly for anything else than documentation and interface.

Definition at line 174 of file default_tuner.h.

MEMBER FUNCTION DOCUMENTATION

INT CONSUMED_ON (CONST TAG & *TAG*) CONST [INLINE]

Tells the scheduler on which process(es) this item is going to be consumed return process id where the item will be consumed (get), or CONSUMER_UNKNOWN (default) or std::vector<int>, containing all ids of consuming processes. To indicate that the consumer processes are unknown, return an empty vector. The vector must contain special values like



CONSUMER_LOCAL. If not CnC::CONSUMER_UKNOWN (or empty vector), this declaration will overwrite what the step-tuner might declare in depends. Providing this method leads to the most efficient communication pattern for to getting the data to were it is needed.

PARAMETERS:

tag	the tag which identifies the item

Definition at line 210 of file default_tuner.h.

```
{
    return CONSUMER_UNKNOWN;
}
```

INT GET_COUNT (CONST TAG & *TAG*) CONST [INLINE]

Allows specifying the number of gets to the given item.

After get_count() many 'get()'s the item can be removed from the collection. By default, the item is not removed until the collection is deleted.

PARAMETERS:

tag	the tag which identifies the item

RETURNS:

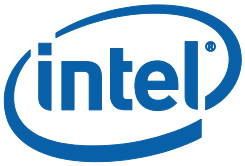
number of expected gets to this item, or CNC::NO_GETCOUNT

Definition at line 194 of file default_tuner.h.

```
{
    return NO_GETCOUNT;
}
```

INT PRODUCED_ON (CONST TAG & *TAG*) CONST [INLINE]

Tells the scheduler on which process(es) this item is going to be produced return process id where the item will be produced. If unknown return CnC::PRODUCER_UNKNOWN. return PRODUCER_LOCAL if local process is the owner. Implementing this method reduces the communication cost for locating and sending data. Implementing item_tuner::consumed_on is more efficient, but might be more complicated. Will be evaluated only if item_tuner::consumed_on returns CnC::CONSUMER_UNKNOWN.



PARAMETERS:

tag	the tag which identifies the item

Definition at line 223 of file default_tuner.h.

```
        {
            return PRODUCER_UNKNOWN;
        }
```

THE DOCUMENTATION FOR THIS STRUCT WAS GENERATED FROM THE FOLLOWING FILE:

- default_tuner.h

PRESERVE_TUNER< TAG, HC > STRUCT TEMPLATE REFERENCE

Inherits tag_tuner< Internal::no_range, default_partitioner<> >.

PUBLIC TYPES

- typedef
- Internal::hash_tag_table< Tag,
- HC > tag_table_type

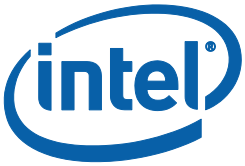
  A tag tuner must provide a tag-table type; default is no tag-table.

PUBLIC MEMBER FUNCTIONS

- bool preserve_tags () const

  return true if tag memoization is wanted; returns false by default (with no_tag_table)

DETAILED DESCRIPTION



TEMPLATE<TYPENAME TAG, TYPENAME HC = CNC_TAG_HASH_COMPARE< TAG >>STRUCT CNC::PRESERVE_TUNER< TAG, HC >

Use this if your tag-collection should preserve tags (memoization) Memoization doesn't work with ranges (yet)

Definition at line 295 of file default_tuner.h.

The documentation for this struct was generated from the following file:

- default_tuner.h

SERIALIZABLE CLASS REFERENCE

DETAILED DESCRIPTION

Definition at line 192 of file serializer.h.

The documentation for this class was generated from the following file:

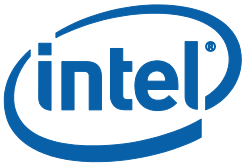
- serializer.h

SERIALIZER CLASS REFERENCE

Handles serilialization of data-objects.

CLASSES

- struct construct_array



- *Allocates an array of type T and size num in pointer variable arrVar.* struct destruct_array

DESTRUCTS THE ARRAY OF TYPE T AND ISZE NUM AT ARRVAR AND RESETS ARRVAR TO NULL.

PUBLIC MEMBER FUNCTIONS

- void resize (size_type len)

  (Re)allocates a buffer of the given size.

- size_type get_header_size () const
- size_type get_body_size () const
- size_type get_total_size () const
- void * get_header () const
- void * get_body () const
- size_type unpack_header () const
- template<class T > serializer & operator& (T &var)
- template<class T > reserved reserve (const T &_obj)
- template<class T > void complete (const reserved &r, const T &_obj)

FRIENDS

- void swap (serializer &s1, serializer &s2)

  Swap internal representations of the given two serializers.

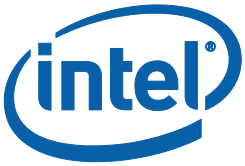
DETAILED DESCRIPTION

Handles serilialization of data-objects.

objects of this class are passed to serialization methods/functions. Try to use operator & only; using anything else is potentially dangerous.

Definition at line 334 of file serializer.h.

MEMBER FUNCTION DOCUMENTATION



VOID COMPLETE (CONST RESERVED & *R*, CONST T & *_OBJ*)

fill in data at position that has been reserved before rserve/complete supported for bitwise-serializable types only

VOID* GET_BODY () CONST [INLINE]

RETURNS:

pointer to the message body (without header)

SIZE_TYPE GET_BODY_SIZE () CONST [INLINE]

RETURNS:

the number of bytes already packed into the buffer

VOID* GET_HEADER () CONST [INLINE]

RETURNS:

pointer to message header

SIZE_TYPE GET_HEADER_SIZE () CONST [INLINE]

RETURNS:

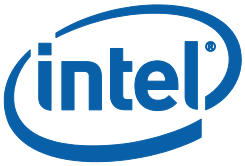
size of header

SIZE_TYPE GET_TOTAL_SIZE () CONST [INLINE]

RETURNS:

total number of bytes in buffer (header + data)

SERIALIZER& OPERATOR& (T & *VAR*)



Top-level packing interface, to be used in the "serialize" function of your classes. Applies the serializer to one object resp. an array of objects (e.g. packing them into the serializer or unpacking them from the serializer). Dispatches w.r.t. the packing category of the object's type (i.e. byte-wise copying or object serialization).

RESERVED RESERVE (CONST T & *_OBJ*)

reserve current position in buffer to be filled later with a call to "complete" rserve/complete supported for bitwise-serializable types only

SIZE_TYPE UNPACK_HEADER () CONST [INLINE]

RETURNS:

body size from header, -1 if something goes wrong

THE DOCUMENTATION FOR THIS CLASS WAS GENERATED FROM THE FOLLOWING FILE:

- serializer.h

STEP_COLLECTION< USERSTEP, TUNER > CLASS TEMPLATE REFERENCE

A step collection is logical set of step instances.

PUBLIC TYPES

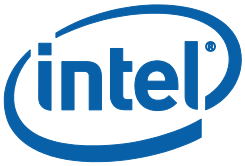
- typedef UserStep step_type

  the type of the step as provided by the user

- typedef Tuner tuner_type

  the type of the tuner as provided by the user

PUBLIC MEMBER FUNCTIONS



- template<typename Derived > step_collection (context< Derived > &ctxt, const std::string &name, const step_type &userStep, const tuner_type &tnr)

  constructor which registers collection with given context

- template<typename DataTag , typename Item , typename ITuner > void consumes (CnC::item_collection< DataTag, Item, ITuner > &)

  Declare this step-collecation as consumer of given item-collection.

- template<typename DataTag , typename Item , typename ITuner > void produces (CnC::item_collection< DataTag, Item, ITuner > &)

  Declare this step-collecation as producer for given item-collection.

- template<typename ControlTag , typename TTuner > void controls (CnC::tag_collection< ControlTag, TTuner > &)

  Declare this step-collecation as controller of given tag-collection.

DETAILED DESCRIPTION

TEMPLATE<TYPENAME USERSTEP, TYPENAME TUNER = STEP_TUNER<>>CLASS CNC::STEP_COLLECTION< USERSTEP, TUNER >

A step collection is logical set of step instances.

A step-collection must be prescribed by a tag-collection and it can be part of consumer/producer relationships with item-collections. Additionally, it can be the controller in control-dependencies (e.g. produce tags).

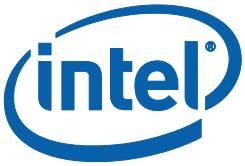
Definition at line 57 of file cnc.h.

CONSTRUCTOR & DESTRUCTOR DOCUMENTATION

STEP_COLLECTION (CONTEXT< DERIVED > & *CTXT*, CONST STD::STRING & *NAME*, CONST STEP_TYPE & *USERSTEP*, CONST TUNER_TYPE & *TNR*)

constructor which registers collection with given context

PARAMETERS:



ctxt	the context this collection belongs to
name	an optional name, used for debug output and tracing
userStep	an optional user step argument, a copy will be created through copy-construction
tnr	an optional tuner object which must persist throughout the lifetime of the step-collection by default a default-constructed tuner will be used.

THE DOCUMENTATION FOR THIS CLASS WAS GENERATED FROM THE FOLLOWING FILE:

- cnc.h

STEP_TUNER< CHECK_DEPS > STRUCT TEMPLATE REFERENCE

Default (NOP) implementations of the tuner interface.

Inherits tuner_base.

PUBLIC MEMBER FUNCTIONS

- template<typename Tag , typename Arg > int priority (const Tag &tag, Arg &arg) const

  Allows defnintion of priorites to individual steps (which are identified by the tag).

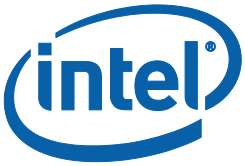
- template<typename Tag , typename Arg , typename T > void depends (const Tag &tag, Arg &arg, T &dC) const

  Allows declaration of data dependencies (to items) of given step (identified by the tag).

- bool preschedule () const

  returns whether the step should be pre-scheduled

- template<typename Tag , typename Arg > int compute_on (const Tag &, Arg &) const



tell the scheduler on which process to run the step (or range of steps) (distCnC)

STATIC PUBLIC ATTRIBUTES

- static const bool check_deps_in_ranges = check_deps

ADDITIONAL INHERITED MEMBERS

DETAILED DESCRIPTION

TEMPLATE<BOOL CHECK_DEPS = TRUE>STRUCT CNC::STEP_TUNER< CHECK_DEPS >

Default (NOP) implementations of the tuner interface.

Also defines the interface a user-provided tuner must satisfy. Derive your tuner from this (to avoid implementing the entire interface).

It is recommended that your tuner does not implement the methods as templates. Instead, you should use the actual types that it expects.

#include <cnc/default_tuner.h>

Definition at line 98 of file default_tuner.h.

MEMBER FUNCTION DOCUMENTATION

INT COMPUTE_ON (CONST TAG & , ARG &) CONST [INLINE]

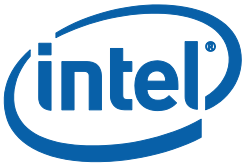
tell the scheduler on which process to run the step (or range of steps) (distCnC)

RETURNS:

process id where the step will be executed, or COMPUTE_ON_ROUND_ROBIN, or COMPUTE_ON_LOCAL, or COMPUTE_ON_ALL, or COMPUTE_ON_ALL_OTHERS

Definition at line 156 of file default_tuner.h.

```
{
    return COMPUTE_ON_ROUND_ROBIN;
}
```



VOID DEPENDS (CONST TAG & *TAG*, ARG & *ARG*, T & *DC*) CONST [INLINE]

Allows declaration of data dependencies (to items) of given step (identified by the tag).

When a step-instance is prescribed through a corresponding tag_collection::put, this method will be called. You can declare dependencies to items by calling dC.depends(item_collection, dependent_item_tag) for every item the step is going to 'get' in its execute method. The actual step execution will be delayed until all dependencies can be satisfied. The default implementation does nothing (NOP). Your own implementation must accept dC by reference (T&). dC.depends accepts an additional optional argument to declare the process that will produce the item. You can either pass the process-rank or CnC::PRODUCER_UKNOWN or CnC::PRODUCER_LOCAL. This producer argument will have effect only if the tuner of the respective item-collection returns CnC::CONSUMER_UKNOWN, otherwise it will be ignored.

PARAMETERS:

tag	the tag which identifies the step to be executed.
arg	the argument as passed to context< Derived >::prescribed (usually the context)
dC	opaque object (must be by reference!) providing method depends to declare item dependencies

Definition at line 132 of file default_tuner.h.

```
        {
        }
```

BOOL PRESCHEDULE () CONST [INLINE]

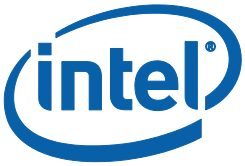
returns whether the step should be pre-scheduled

Pre-scheduling provides an alternative method for detecting data dependencies.

The step instance will be run immediately when prescribed by a tag_collection::put. All items that are not yet available when accessed by the non-blocking get method will automatically be treated as dependent items. The pre-run will end at context::flush_gets() if any items are unavailable. The step execution will be delayed until all detected dependencies can be satisfied.

Definition at line 148 of file default_tuner.h.

```
        {
            return false;
        }
```



INT PRIORITY (CONST TAG & *TAG*, ARG & *ARG*) CONST [INLINE]

Allows defnintion of priorites to individual steps (which are identified by the tag).

RETURNS:

the default implementation always return 1.

PARAMETERS:

tag	the tag which identifies the step to be executed
arg	the argument as passed to context< Derived >::prescribed (usually the context)

Definition at line 105 of file default_tuner.h.

```
        {
            return 1;
        }
```

MEMBER DATA DOCUMENTATION

CONST BOOL CHECK_DEPS_IN_RANGES = CHECK_DEPS [STATIC]

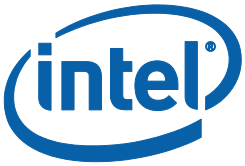
true if steps launched through ranges consume items or need global locking, false otherwise. Avoiding checks for dependencies and global locks saves overhead and will perform better (e.g. for parallel_for). Safe execution (with checks) is the default (check_deps template argument).

Definition at line 164 of file default_tuner.h.

THE DOCUMENTATION FOR THIS STRUCT WAS GENERATED FROM THE FOLLOWING FILE:

- default_tuner.h

ITEM_TUNER< TT >::TABLE_TYPE< TAG, ITEM > STRUCT TEMPLATE REFERENCE



Defines the type of the internal data store.

DETAILED DESCRIPTION

TEMPLATE<TEMPLATE< TYPENAME T, TYPENAME I > CLASS TT>TEMPLATE<TYPENAME TAG, TYPENAME ITEM>STRUCT CNC::ITEM_TUNER< TT >::TABLE_TYPE< TAG, ITEM >

Defines the type of the internal data store.

It forwards the functionality of the template parameter template. The expected interface of the template is not yet exposed. Use hashmap_tuner or vector_tuner to derive you own tuner.

Definition at line 182 of file default_tuner.h.

The documentation for this struct was generated from the following file:

- default_tuner.h

TAG_COLLECTION< TAG, TUNER > CLASS TEMPLATE REFERENCE

A tag collection is a set of tags of the same type. It is used to prescribe steps. By default, tags are not stored.

PUBLIC TYPES

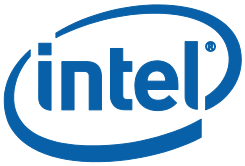
- typedef Tag tag_type

  the tag type

PUBLIC MEMBER FUNCTIONS

- template<class Derived > tag_collection (context< Derived > &ctxt, const std::string &name, const Tuner &tnr)

  constructor which registers collection with given context



- template<typename UserStep , typename STuner , typename Arg > error_type prescribes (const step_collection< UserStep, STuner > &s, Arg &arg)

  Declare the prescription relationship between the tag collection and a step collection.

- void put (const Tag &t)

  prescribe the associated step. If we are preserving tags for this collection, make a copy of the tag and store it in the collection.

- void put_range (const typename Tuner::range_type &r)

  prescribe an entire range of tags

- void reset ()

  removes all of the tag instances from the collection

- size_t size ()

  returns number of elements in collection

- bool empty ()

  returns true if size()==0, false otherwise

DETAILED DESCRIPTION

TEMPLATE<TYPENAME TAG, TYPENAME TUNER = TAG_TUNER<>>CLASS CNC::TAG_COLLECTION< TAG, TUNER >

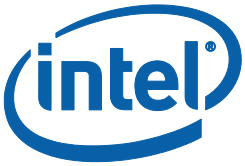
A tag collection is a set of tags of the same type. It is used to prescribe steps. By default, tags are not stored.

Tag must provide copy and default constructors and the assigment operator.

If Tag is not convertable into size_t, a suitable hash_compare class must be provided which satisifies the requirements for tbb::concurrent_hash_map. The default cnc_tag_hash_compare works for types that can be converted to size_t and have an operator==. You can provide a specialized template for cnc_tag_hash_compare or specify and implement your own compatible class.

Definition at line 115 of file cnc.h.

CONSTRUCTOR & DESTRUCTOR DOCUMENTATION



TAG_COLLECTION (CONTEXT< DERIVED > & *CTXT*, CONST STD::STRING & *NAME*, CONST TUNER & *TNR*)

constructor which registers collection with given context

PARAMETERS:

ctxt	the context this collection belongs to
name	an optional name, used for debug output and tracing
tnr	an optional tuner object which must persist throughout the lifetime of the tag-collection by default a default-constructed tuner will be used.

MEMBER FUNCTION DOCUMENTATION

ERROR_TYPE PRESCRIBES (CONST STEP_COLLECTION< USERSTEP, STUNER > & *S*, ARG & *ARG*)

Declare the prescription relationship between the tag collection and a step collection.

```
    \param s class representing step collection. s is required to
    provide the following const method, where Arg a is the optional
    parameter described below.
      @code
```

int execute(const Tag & tag, Arg & a) const; A copy of s will be created by calling its copy constructor.

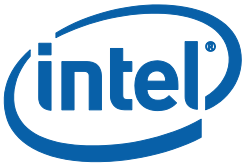
PARAMETERS:

arg	This argument will be the parameter passed to Step::execute and the tuner methods. The object must exist as long as instances of the given step might be executed. Usually arg will be the containing context.

RETURNS:

0 if succeeded, error code otherwise

VOID PUT (CONST TAG & *T*)



prescribe the associated step. If we are preserving tags for this collection, make a copy of the tag and store it in the collection.

PARAMETERS:

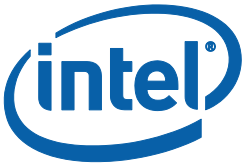
t	the tag to be put

VOID PUT_RANGE (CONST TYPENAME TUNER::RANGE_TYPE & *R*)

prescribe an entire range of tags

PARAMETERS:

r	A range, which is potentially splittible through a partitioner. Following the TBB range/splittable concept, extended by STL container requirements, a range R must provide the following interface: • R::R(const R&) : Copy constructor. • int size() : return number of elements (tags) in range • const_iterator : forward iterator (operator++, operator tag_type() const) to make it work with tbb::blocked_range, the cast operator is used instead of operator*(). • const_iterator begin() const : first member of range • const_iterator end() const : Exclusive upper bound on range Using it with the default_partitioner also requires • R::R(R& r, tbb::split) : Split r into two subranges. • bool R::is_divisible() const : true if range can be partitioned into two subranges.



THE DOCUMENTATION FOR THIS CLASS WAS GENERATED FROM THE FOLLOWING FILE:

- cnc.h

TAG_PARTITIONER< GRAINSIZE > CLASS TEMPLATE REFERENCE

Inherits default_partitioner< grainSize >.

PUBLIC TYPES

- typedef range_is_tag_type split_type

PUBLIC MEMBER FUNCTIONS

- template<typename Range , typename StepInstance > bool divide_and_originate (Range &range, StepInstance &si) const

  divide given range into in arbitrary number of ranges of type Range

ADDITIONAL INHERITED MEMBERS

DETAILED DESCRIPTION

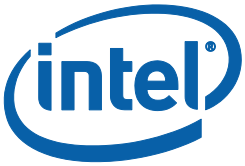
TEMPLATE<INT GRAINSIZE = 0>CLASS CNC::TAG_PARTITIONER< GRAINSIZE >

Use this instead of default_partitioner if your tag is self-dividing (e.g. a range) and you want to use the partitioning mechanism through cnC::tag_collection::put_range

Definition at line 172 of file default_partitioner.h.

MEMBER TYPEDEF DOCUMENTATION

TYPEDEF RANGE_IS_TAG_TYPE SPLIT_TYPE



set to range_is_tag_type if tag is self-dividing, e.g. if the range-type is also the tag-type as passed to the step
set to range_is_range_type if tag is undivisible, e.g. if range-type != step_type

Reimplemented from default_partitioner< grainSize >.

Definition at line 177 of file default_partitioner.h.

MEMBER FUNCTION DOCUMENTATION

BOOL DIVIDE_AND_ORIGINATE (RANGE & *RANGE*, STEPINSTANCE & *SI*) CONST [INLINE]

divide given range into in arbitrary number of ranges of type Range

Call si.originate_range(r) for each new range. The original - but modified - range must *not* be passed to originate_range! If tag-types are self-dividing (e.g. if range-type == tag-type) you should call "originate" instead of "originate_range" for leaves of the recursive range-tree.

RETURNS:

true if the orignal - but modified - range needs further splitting, false if no further splitting is desired.

The aggregated set of the members of the sub-ranges applied to "t.originate[_range]" must equal the set of member in given range. Overlapping ranges or gaps may lead to arbitrary effects.

PARAMETERS:

range	the original range to split, may be altered
si	opaque object, call t.originate[_range](r) for all split-off sub-ranges

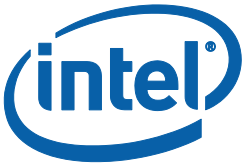
Reimplemented from default_partitioner< grainSize >.

Definition at line 205 of file default_partitioner.h.

```
    {
        return this->divide_and_originate( range, si, this->grain_size(
range.size() ) );
    }
```

THE DOCUMENTATION FOR THIS CLASS WAS GENERATED FROM THE FOLLOWING FILE:

- default_partitioner.h



TAG_TUNER< RANGE, PARTITIONER > STRUCT TEMPLATE REFERENCE

Inherits tuner_base.

PUBLIC TYPES

- typedef Range range_type

  A tag tuner must provide the type of the range, default is no range.

- typedef Internal::no_tag_table tag_table_type

  A tag tuner must provide a tag-table type; default is no tag-table.

- typedef Partitioner partitioner_type

  The type of the partitioner.

PUBLIC MEMBER FUNCTIONS

- partitioner_type partitioner () const

  return a partitioner for range-based features, such as parallel_for

- bool preserve_tags () const

  return true if tag memoization is wanted; returns false by default (with no_tag_table)

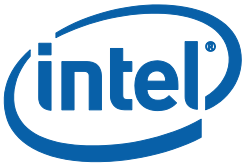
ADDITIONAL INHERITED MEMBERS

DETAILED DESCRIPTION

TEMPLATE<TYPENAME RANGE = INTERNAL::NO_RANGE, TYPENAME PARTITIONER = DEFAULT_PARTITIONER<>>STRUCT CNC::TAG_TUNER< RANGE, PARTITIONER >

Default implementations of the tag-tuner interface for tag-collections Use this if you are going put ranges. Optional argument is a custom partitioner. Ranges don't work with memoization (yet)

Definition at line 265 of file default_tuner.h.



MEMBER FUNCTION DOCUMENTATION

PARTITIONER_TYPE PARTITIONER () CONST [INLINE]

return a partitioner for range-based features, such as parallel_for

SEE ALSO:

default_partitioner for the expected signature of partitioners overwrite partitioner() if it doesn't come with default-constructor or if the default constructor is insufficient.

Definition at line 278 of file default_tuner.h.

```
{
    return typename tag_tuner< Range, Partitioner >::partitioner_type();
}
```

THE DOCUMENTATION FOR THIS STRUCT WAS GENERATED FROM THE FOLLOWING FILE:

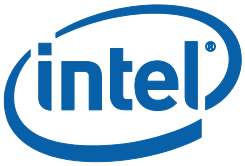
- default_tuner.h

TUNER_BASE CLASS REFERENCE

Inherited by item_tuner< TT > [virtual], step_tuner< check_deps > [virtual], tag_tuner< Range, Partitioner > [virtual], item_tuner< Internal::hash_item_table > [virtual], item_tuner< Internal::vec_item_table > [virtual], and tag_tuner< Internal::no_range, default_partitioner<> > [virtual].

STATIC PUBLIC MEMBER FUNCTIONS

- static int myPid ()



- static int numProcs ()

DETAILED DESCRIPTION

Functionality that might be needed to implement all kinds of tuners. Always try to use higher level tuners which derive from this. Always use virtual inheritance (see higher leve tuners).

Definition at line 67 of file default_tuner.h.

MEMBER FUNCTION DOCUMENTATION

STATIC INT MYPID () [INLINE], [STATIC]

returns id/rank of calling process defaults to 0 if running on one process only.

Definition at line 72 of file default_tuner.h.

```
        {
            return Internal::distributor::myPid();
        }
```

STATIC INT NUMPROCS () [INLINE], [STATIC]

return total number of processes participating in this programm execution defaults to 1 if running on one process only.

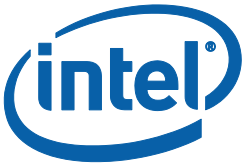
Definition at line 78 of file default_tuner.h.

```
        {
            return Internal::distributor::numProcs();
        }
```

THE DOCUMENTATION FOR THIS CLASS WAS GENERATED FROM THE FOLLOWING FILE:

- default_tuner.h

VECTOR_TUNER STRUCT REFERENCE



The tuner base for vector-based item-tuners.

Inherits item_tuner< Internal::vec_item_table >.

ADDITIONAL INHERITED MEMBERS

DETAILED DESCRIPTION

The tuner base for vector-based item-tuners.

Your tags must be convertable to and from size_t. You must provide the maximum value before accessing the collection (constructor or set_max). The runtime will allocate as many slots. Hence, use this only if your tags-space is dense, without a large offset and if it is not too large.

Definition at line 254 of file default_tuner.h.

The documentation for this struct was generated from the following file:

- default_tuner.h

INDEX

INDEX