Intel® Concurrent Collections for C++ In Action

2012/12/06

4th Annual Concurrent Collections Workshop

Frank Schlimbach
Kath Knobe
James Brodman
Goal of this tutorial

The big idea

• You should learn
  – How to design a CnC program
  – How to write a CnC program in C++
  – How nice this all is

• We would like to learn
  – What’s difficult to understand
  – What helps with understanding
  – What application you might have to use CnC
Action Overview

• Why CnC and what’s CnC all about?
• Let’s Cholesky
  – Basic
  – Ready for distributed memory
• Tuning
  – Get-counting
  – Pre-declaring dependencies
• CnC loves to share pointers
• Tuning for distributed memory
• CnC in the cloud
## Cholesky

- LU factorization
- Expects **symmetric** positive-definite matrix
- Works on lower triangular matrix
- Blocked algorithm
- In each iteration $k$
  - Operates on diagonal element $(k,k)$ and its lower triangular submatrix
  - Computes final value for column $k$

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$(1,1)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$(2,2)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(3,3)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(4,4)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$(5,5)$</td>
</tr>
</tbody>
</table>
Cholesky (5x5 blocks) : Iteration k=0

```c
for( int k = 0, k < N; ++k ) {
    cholesky( tiles, k );
    for( int j = k+1; j < N; ++j ) {
        trisolve( tiles, j, k );
        for( int i = k+1; i <= j; ++i ) {
            update( tiles, i, j, k );
        }
    }
}
```
Cholesky (5x5 blocks) : Iteration k=1

```c
for( int k = 0, k < N; ++k ) {
    cholesky( tiles, k );
    for( int j = k+1; j < N; ++j ) {
        trisolve( tiles, j, k );
        for( int i = k+1; i <= j; ++i ) {
            update( tiles, i, j, k );
        }
    }
}
```
for( int k = 0, k < N; ++k ) {
    cholesky( tiles, k );
    for( int j = k+1; j < N; ++j ) {
        trisolve( tiles, j, k );
        for( int i = k+1; i <= j; ++i ) {
            update( tiles, i, j, k );
        }
    }
}
for( int k = 0, k < N; ++k ) {
    cholesky( tiles, k );
    for( int j = k+1; j < N; ++j ) {
        trisolve( tiles, j, k );
        for( int i = k+1; i <= j; ++i ) {
            update( tiles, i, j, k );
        }
    }
}
Cholesky (5x5 blocks) : Iteration k=4

```
for( int k = 0, k < N; ++k ) {
    cholesky( tiles, k );
    for( int j = k+1; j < N; ++j ) {
        trisolve( tiles, j, k );
        for( int i = k+1; i <= j; ++i ) {
            update( tiles, i, j, k );
        }
    }
}
```
3 steps per iteration

1. **Cholesky**
   - **Sequential**
   - **Reads output from previous iteration (k-1)**

2. **Trisolve**

3. **Update**
3 steps per iteration

1. Cholesky
   - sequential

2. Trisolve
   - Reads outputs from cholesky\((k)\) and previous iteration update\((k-1)\)
   - Each row can go in parallel for given column \(k\)

3. Update
3 steps per iteration

1. Cholesky
   - sequential
2. Trisolve
3. Update
   - Reads 2 outputs from trisolve\((k)\) and 1 from previous iteration update\((k-1)\)
   - All blocks can go in parallel
   - Diagonal reads only 2 inputs!
3 steps per iteration

1. Cholesky
   - sequential

2. Trisolve

3. Update
   - Reads 2 outputs from trisolve(k) and 1 from previous iteration update(k-1)
   - All blocks can go in parallel
   - Diagonal reads only 2 inputs!
### Cholesky

Next iteration reads output from update.
Cholesky: available parallelism

• Intra-iteration
  – Row-parallelism (trisolve)
    – Need to sync with cholesky
  – Block-parallelism (update, update&trisolve)
    – Need to sync with trisolve

• Inter-iteration
  – cholesky(k) can go in parallel with
    – Trisolve(k-1,[k+2..n])
    – Update(k-1,[k+2..n],[k+2..n])
    – ...

![Diagram of parallelism](image)
Explicit parallelism

```c
for( int k = 0, k < N; ++k ) {
    cholesky( matrix, k );
    for( int j = k+1; j < N; ++j ) {
        trisolve( array, j, k );
        for( int i = k+1; i <= j; ++i ) {
            update( array, i, j, k );
        }
    }
}
```

• Simple for-all loops don’t work here
• Only in the innermost loop
• Could split loops and add barriers
  • Barriers don’t scale
  • Still not utilizing a lot of the available parallelism
• ...?
Explicit Parallelism Is Hard

What makes it hard?

- Having to think about algorithm and parallelization at the same time
- Requires understanding of the target platform
- Thinking parallel is difficult and error-prone

Pipeline?
Data Parallel?
Tasking?
Parallel_for?
Mutex?
Barrier?
xx_allgatherv?
Future?
xx_spawn?
Sync?
...

My algorithm
Couldn’t parallelism be easy?

• Why not just say where the dependences are between computations?
• Everything else could go in parallel
• As the application programmer I know the dependences, but thinking about the inverse is hard
• E.g. why not just say
  – Trisolve(iter, col)
    needs Tile( iter, col, col )
    needs Tile( iter-1, col, row )
    compute()
    produces Tile( iter, col, row )
  – Etc.
Yes! That’s what CnC does!

What makes it hard?

🛠️ Having to think about algorithm and parallelization at the same time
🛠️ Requires understanding of the target platform
🛠️ Thinking parallel is difficult and error-prone

Pipeline?
Data Parallel?
Tasking?
Parallel_for?
Mutex?
Barrier?
xx_allgatherv?
Future?
xx_spawn?
Sync?
...

Runtime
The big idea

Specify what operations need to be in order

Easy, known and depends only on application

Don’t specify how operations run in parallel

Difficult and depends on target
A Different Approach to Parallelism

Domain
Need to be understood even for a sequential program

What are the semantic dependencies of my algorithm?

Can the scheduling be improved?

Day 1
Thinking about the algorithm (only)

Day X (if needed)
Thinking about tuning (only)

Tuning

Let the compiler and the do the complicated work.
Exactly Two Ordering Constraints

Constraints on the execution order define what computation units need to be sequential.

Exactly two of them exist.
Exactly Two Ordering Constraints: 1. data

- Producer must execute before consumer
Exactly Two Ordering Constraints: 2. control

- Producer must execute before consumer
- Controller must execute before controlee
Graphs define CnC applications

- In CnC you design your application as a graph built with these relationships
  - They state if and which ordering relations exist between compute units
  - Control- and Data-collections act as intermediary
  - Describes data-flow and control-flow
- CnC is a control language
  - The runtime controls the execution of compute units to enforce the specified ordering requirements
- If no ordering constraint exists, there is potential parallelism!
The CnC Graph is a high-level design

- The graph nodes are placeholders/collections for potentially many instances of the same type

Step-instances execute stateless computation
The collections hold the execution instances

Data/item-instances are uniquely identified by a key/tag
The collections map the unique keys to values (immutable data)

Control/tag-instances identify which related/prescribed step-instances are to be executed
The collections hold the control instances

When a producer step (-instance) executes it will write/create one or more of these data-items (or even none)

When a consumer step (-instance) executes it will read one or more of these data-items (or even none)

The controller-step emits the "control-point"

Each tag/control (-instance) prescribes an execution of the related step
Classic tree walk

a node in the tree

```cpp
myStep<nodeID> {  
    get nodeContents[ nodeID ]  
    Compute( nodeID )  
}
```
Classic tree walk from node to its children

```c
myStep<nodeID> {
    get nodeContents[ nodeID ]
    Compute( nodeID )
    put[rightChild]
    put[leftChild]
    if not_done_yet
        maybe putControl[right>
        maybe putControl[left>
    }
```
Classic tree walk initialize root

```java
myStep<nodeID> {
    get nodeContents[ nodeID ]
    Compute( nodeID )
    put[rightChild]
    put[leftChild]
    if not_done_yet
        maybe putControl<right>
        maybe putControl<left>
}
```
Classic tree walk initialize root

myStep<nodeID> {
    get nodeContents[ nodeID ]
    Compute( nodeID )
    put[rightChild]
    put[leftChild]
    if not_done_yet
        maybe putControl[right]
        maybe putControl[left]
}

Breadth-first vs depth-first is a scheduling/tuning issue. This domain spec is valid for both.
CnC High-level summary

- A CnC graph explicitly expresses the semantically required ordering constraints
  - Provides a global view of the application
- This implicitly exposes the available parallelism
  - No „artifical“ barriers or alike
- CnC defines rules
  - Dynamic Single Assignment
  - Stateless and side-effect-free steps
- It is easier to follow CnC rules than to make parallelism explicit and avoid race conditions
- Following the rules gives guarantees
  - Determinism with respect to result
  - Independence of #threads/#processes
  - Moving from shared to distributed memory is simple
Questions?

We’ll write a cholesky algorithm with CnC which
- Expresses no explicit parallelism
- Exposes all available parallelism at runtime

We’ll also do some tuning and run the application on distributed memory.
Concurrent Collections (CnC)

CnC is a higher-level programming model.

Intel® Concurrent Collections for C++ provides the infrastructure to develop software in C++ in the Concurrent Collections programming model.

Other implementations exist elsewhere (Java, C, Haskell,...)
Installation

- Download CnC kit, (unpack) and install
- Linux setup
  - CnC
    csh: source <cnc-v0.8>/bin/intel64/cncvars.csh
    bash: source <cnc-v0.8>/bin/intel64/cncvars.sh
  - Manual setup of CnC
    - Translator and binaries use shared libs
    - $LD_LIBRARY_PATH should contain <cnc-v0.8>/lib/intel64
    - CnC/Intel convention in Makefiles use
      - $CNCROOT
- Windows setup
  - Install
  - Start VS10 *after* installation completes
  - Might need to logout/login

TBB >= 4.0 needs to be installed
Common problems and sanity checks

• Try building/running samples in `<cnc-v0.8>/samples`

• Having trouble?
  – Did you modify and source `tbbvars.sh` & `cncvars.sh`

  – Sanity check: run `env | grep -i tbb`
    – Do you see something in `LD_LIBRARY_PATH`

  – Sanity check: run `env | grep -i cnc`
    – Do you see something in `LD_LIBRARY_PATH`
Intel® Concurrent Collections for C++

• Template Library for C++
  – For Windows IA-32 and Intel-64, Linux Intel-64
  – Standard C++, works with common compilers (e.g. GNU*, Intel, Visual Studio*)

• Shared and distributed memory

• Documentation

• Samples

Intel® Concurrent Collections: Architecture

```
CnC C++ API

CnC runtime

TBB

libc  pthread  ...

OS

MPI
```
How to write a CnC application

• Design
  – 4 phase methodology

• Write C++ code
  – The spec/graph
    – C++ API lets you define collections and wire them
  – The steps
    – Each step is a user function with required interface
    – Steps consume and produce item/tag instances with `get` and `put`
  – The environment
    – produces initial tag/item instances and invokes the graph
Designing with CnC

- Four phase methodology for creating a CnC graph
- Result is a “modular” system
  - High degree of independence between compute kernels
- Mapping to C++ becomes trivial
1: The White Board Drawing

- computations
- data
- producer/consumer relations
- I/O
2: Distinguish among the instances

for( int k = 0, k < N; ++k ) {
    cholesky( tiles, k );
    for( int j = k+1; j < N; ++j ) {
        trisolve( tiles, j, k );
        for( int i = k+1; i <= j; ++i ) {
            update( tiles, i, j, k );
        }
    }
}
3: What are the control tag collections

```c
for( int k = 0, k < N; ++k ) {
    cholesky( tiles, k );
    for( int j = k+1; j < N; ++j ) {
        trisolve( tiles, j, k );
        for( int i = k+1; i <= j; ++i ) {
            update( tiles, i, j, k );
        }
    }
}
```
4: Who produces control

- **CONTROL TAG**: CholeskyTag: iter
  - **COMPUTE STEP**: Cholesky: iter
- **CONTROL TAG**: TrisolveTag: row, iter
  - **COMPUTE STEP**: Trisolve: row, iter
- **CONTROL TAG**: UpdateTag: col, row, iter
  - **COMPUTE STEP**: Update: col, row, iter
- **DATA ITEM**: Tiles: col, row, iter
Tutorial

• Copy tutorial material

• Each exercise comes in a separate directory
  – File name/path given at bottom of slide
  – Places to be filled are marked with “****”
  – cholesky.h: CnC specification
  – cholesky_cnc.cpp: step-code and environment
  – cholesky_types.h: typedefs
  – cholesky.cpp: driver/command-line parsing
  – Makefile

• All come with a starting point
  – Initial code
  – Solution in sub-directory
Let’s start with a few type-defs

```cpp
// we use int for single index 'k'

// we use a pair for (k,j)
typedef std::pair<int,int> pair;

// we use our own class for the triple (k,j,i)
class triple : public std::array<int,3>
{
    public:
        triple( int a = 0, int b = 0, int c = 0 ) { (*this)[0] = a; (*this)[1] = b; (*this)[2] = c; }
};

// Our Tile; its size is defined at runtime
template< typename T >
class Tile
{
    public:
        Tile( int sz = 0 ) : m_sz( sz ), m_array( sz*sz ) {}
        Tile( const Tile< T > & o ) : m_sz( o.m_sz ), m_array( o.m_array ) {}
        #define TOI( _i, _j, _s ) ((_j)*(_s)+(_i))
        inline T operator()( int i, int j ) const { return m_array[TOI(i,j,m_sz)]; }
        inline T & operator()( int i, int j ) { return m_array[TOI(i,j,m_sz)]; }
    private:
        std::vector< T > m_array;
        int m_sz;
};
```

Types of our tags (for control and data)

Type of our data
CnC Cholesky (reminder)

- **CONTROL TAG**: CholeskyTag: iter
  - **COMPUTE STEP**: Cholesky: iter

- **CONTROL TAG**: TrisolveTag: row, iter
  - **COMPUTE STEP**: Trisolve: row, iter

- **CONTROL TAG**: UpdateTag: col, row, iter
  - **COMPUTE STEP**: Update: col, row, iter

- **DATA ITEM**: Tiles: col, row, iter
3 compute kernels/steps

Steps are stateless and have no side-effects

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

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

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

Tags used to
• Identify data
• Parametrize work/step

cholesky_basic/cholesky.h
Defining the graph in the context

```cpp
struct cholesky_context
{
    public CnC::context< cholesky_context >
    {
        // Step Collections
        CnC::step_collection< cholesky_step > cholesky;
        CnC::step_collection< trisolve_step > trisolve;
        CnC::s********** update;
        // Item collections
        CnC::item_collection< *, tile_type > tiles;
        // Control collections
        CnC::tag_collection< int > control_cholesky;
        CnC::t********** control_trisolve;
        CnC::tag_collection< triple > control_update;
        ...
    }
};
```

 CONTROL TAG
CholeskyTag: iter

 CONTROL TAG
TrisolveTag: row, iter

 CONTROL TAG
UpdateTag: col, row, iter

 DATA ITEM
Tiles: col, row, iter

step-type

itemtag-type

data-type

controltag-type == steptag-type

cholesky_basic/cholesky.h
struct cholesky_context
  : public CnC::context< cholesky_context >
{
  ...
  // The context class constructor wires the collections
  cholesky_context( int _b = 0, int _p = 0, int _n = 0 )
    : $initializer-list$
  {
    // Prescriptive relations
    control_cholesky.prescribes( cholesky, *this );
    control_trisolve.prescribes( trisolve, *this );
    control_update.p***************************;
    // no control relations between steps
    // producer/consumer relations
    cholesky.consumes( tiles );
    cholesky.produces( tiles );
    trisolve.consumes( tiles );
    trisolve.produces( tiles );
    update.c***************************;
    update.p***************************;
  }
};

cholesky_basic/cholesky.h
Step update

// Performs symmetric rank-k update of the submatrix.
// Input to this step is the given submatrix and the output of the previous step.
int update_step::execute( const triple & t, cholesky_context & c ) const
{
    const int b = c.b; const int k = t[0]; const int j = t[1]; const int i = t[2];
    assert( j != k && i != k );

    tile_type A_block, L1_block, L2_block;
    c.tiles.get( triple( k, j, i ), A_block ); // Get the input tile.

    if( i==j ){ // Diagonal tile.
        c.tiles.get( *****************, ********** ); // both the tiles are the same.
    } else{ // Non-diagonal tile.
        c.tiles.get( triple( k+1, i, k ), L2_block ); // Get the first tile.
        c.tiles.get( triple( k+1, j, k ), L1_block ); // Get the second tile.
    }

    // A_block is a copy, a local variable, so we can overwrite at will
    for( int j_b = 0; j_b < b; j_b++ ) {
        for( int k_b = 0; k_b < b; k_b++ ) {
            A_block( i_b, j_b ) = ...;
        }
    }

    c.tiles.put( *****************, A_block ); // Write the output at the next iteration.
    return CnC::CNC_Success;
}
Getting it started: I/O

- Ok, fine, but where’s the trigger?
- We need control!
  - (no control in the cholesky graph, it is known “statically”)
- For execution, the CnC graph needs input
  - data instances
  - control instances
- It must come from the outside (main, environment)
  1. create the graph
  2. Put control and data
  3. [wait]
  4. Get results
// here's our "main" for doing cholesky on a given input matrix
void cholesky( double * A, const int n, const int b, const char * oname )
{
    // Create an instance of the context class which defines the graph
    cholesky_context c( b, p, n );
    // loop for putting data and control
    for( int k = 0; k < p; k++ ) {
        // First let's create and put all the input tiles
        int i = k;
        for( int j = 0; j <= i; j++ ) {
            c.tiles.put( triple( 0, i, j ), extract( A, j, i ) );
        }
        // Now let's put control tags, we know all of them "statically"
        c.control_cholesky.put( k );
        for( int j = k+1; j < p; j++ ) {
            c.control_trisolve.put( pair( k, j ) );
            for( int i = k+1; i <= j; i++ ) {
                c.control_update.put( triple( k, j, i ) );
            }
        }
    }
    // Now just wait for all steps to finish
    c.wait( );
    // Now get the output tiles and print
    for( i = ... ) {
        for( j = ... ) {
            tile_type _tmp;
            c.tiles.get( triple( i, j, i ), _tmp );
            pretty_print( _tmp );
        }
    }
}
Compile and Run

- See Makefile for compiler/linker flags
  ```shell
  make cholesky
  ```
- Run on a 1000x1000 matrix and tilesize 100
  ```shell
  ./cholesky 1000 100
  ```

Trouble?
- Have you sourced cncvars.[c]sh?
- Have you sourced tbbvars.[c]sh intel64?
How do I know it’s doing the right thing?

• Let’s break the code and see what happens
  – Change cholesky_cnc.cpp:174 to put only tags with odd i

  ```
  c.control_update.put( triple( k, j, i ) );
  -> if( i%2 ) c.control_update.put( triple( k, j, i ) );
  ```

• It shouldn’t be able to complete
  – Make & run

• When it finds no more work to do, it’ll report about prescribed steps which are waiting for their input
  – They are waiting for the input of the not-prescribed update steps
Tracing

• Built-in tracing capability prints event like
  – item or tag put
  – item get
  – Step launched, finished re-scheduled

• Enabled per collection or for everything

  • CnC::debug::trace( <collection> );
  • CnC::debug::trace_all( <context> );

• Demo
Control and scheduling

• When a tag is put, we say that an step-instance for each prescribed step-collection should be executed ("whether")
• The instances are handed to a low-level task-scheduler (selectable at runtime)
• Scheduler can decide about the "when" to execute

```
[1]> ./cholesky 1000 100
The time taken for parallel execution a matrix of size 1000 x 1000 : 0.0709729 sec
Steps created( 220 )
Steps scheduled( 607 ) inflight( 0, 0 )
```
Data dependencies and re-scheduling

- The runtime implements a abort-and-retry strategy
- if items are unavailable
  -> abort and put aside
- If item is put, re-schedule aborted step

1. Step execute
2. get()
3. Available ?
   yes
   Return item
   no
5. Register step-inst. with item
6. Get new work
7. Throw exception

- Get
- Register
- Execute
- Return
- Available
- Throw
Scheduling strategies

- Other scheduling strategies are possible
- Different approaches have been implemented in different implementations
- With more information, smarter decisions can be made
Asynchronous Parallelism

Execution of an iterative stencil code

- x-axis: time
- Y-axis: number of threads
- Color: iteration number

Synchronous „parallel_for“

Asynchronous with CnC

One iteration at a time

Iterations overlap
Rules for implementing steps

• Do all the gets first (if possible)
• Protect prior memory allocation against exceptions
• Don’t use global variables
  – There is a “blessed” way to use global DSA data
• Never ever write to global data
• Think stateless
  – step-instances die when done, no way to keep state between them
  – They can share a global immutable “state” within a collection
Runtime Options

• CnC runtime has switches used through environment variables
  – CNC_NUM_THREADS sets the number of threads
  – CNC_SCHEDULER sets the scheduling policy
  – DIST_CNC sets the communication layer
Debugging and Profiling

• Straightforward debugging, no magic mechanisms
  – Use common techniques
  – E.g. totalview with MPI
  – E.g. gdb/idb/Visual Studio with sockets

• Macro CNC_USE_ASSERT

• Hooks for profiling with Intel® Trace Analyzer and Collector
  – Standard use with MPI
  – Built-in hooks in socket runtime and for local-only
  – Convenient macro for manual instrumentation of user code
Why distributed CnC?

• CnC is a declarative and high level model
  – Abstracts from memory model
  – Facilitates switch between shared and distributed memory
    (no explicit message passing needed)

• CnC comes with a control methodology
  – Allows controlling work distribution

• CnC comes with a data methodology
  – Provides hooks for selective data distribution

CnC provides a unified programming model for shared and distributed memory.
Making a CnC program distCnC-ready*

• `#include <cnc/dist_cnc.h>`
  - sets `#define` and declares `dist_cnc_init` template

• instantiate `CnC::dist_cnc_init< ... >` object
  - First thing in `main`, must persist throughout `main`
  - Template parameters are the contexts used in the program

• serialization of non-standard data types (tags and items)
  - Simple mechanism (similar to BOOST)
  - int, double, float, char etc. don’t need explicit serialization

• **Same binary runs on shared and distributed memory**

*distCnC-readyness doesn’t guarantee good performance, but it enables execution on a distributed memory system.*
Why Serialization

• Distributed memory systems require serialization for data transfer
  ⇒ Tags and items must be serializable
• C++ language does not provide serialization (like Java or .NET)
• CnC framework provides serialization capabilities which
  ➢ Make simple things simple
    ⇒ Built-in serialization of standard data types and ranges
    ⇒ Array-wrappers with and without memory handling
  ➢ Make complex things possible
    ⇒ All data types can be serialized
    ⇒ Complex structures (e.g. with pointers or virtual methods) require `serialize` method or function
  ➢ Are easy to use and commonly known (like in Boost)
  ➢ Do not provide automatism which might fail
    ⇒ auto-serialization only upon request (simple declaration) compiler issues error if serialization is undeclared
Serialization

**Bitwise serializable** (e.g. structs without pointers; default for builtin types)

```cpp
CNC_BITWISE_SERIALIZABLE( MyStruct )
```

**Explicitly serializable** (default)

provide

```cpp
void serialize( CnC::serializer & , YourType & )
```
or

```cpp
void YourType::serialize( CnC::serializer & )
```

one function/method for both serialization and deserialization

very easy syntax, using `operator&` (like in Boost)

```cpp
class MyType {
    int _n;
    float* __arr;
    MySubClass _obj;
public:
    void serialize( CnC::serializer & buf ) { 
        buf & _n;          // standard data type
        buf & CnC::chunk< float >( _arr, _n ) // automatic allocation
        & _obj;            // requires its serializability
    }
};
```
Things to keep in mind

• Collections must be members of contexts (constructed in its ctor)
• Context must be default constructible and prescribe steps there
• Tags must not be of pointer type
  – Data (items) can well be pointers
• Global variables are evil and must not be used (within the execution scope of steps)
• In contrast to local-only execution, preservation of steps will only locally suppress redundant step execution.

• All this is aligned with CnC’s methodology!
Limitations of distributed computing apply

• Usual caveats for distributed memory apply
  – E.g. ratio between data-exchange and computation

• Different algorithms might be needed for distributed or shared memory
  + Programming methodology and framework stays the same in any case
  + Over a wide class of applications the algorithm stays the same
Dist-ready 1 + 2

1. In cholesky_types.h
   ...
   #ifdef _DIST_
   #include <cnc/dist_cnc.h>
   #else
   #include <cnc/cnc.h>
   #endif

2. In cholesky.cpp
   ...
   int main( int argc, char * argv[] )
   {
      #ifdef _DIST_
      CnC::dist_cnc_init< *********** > _dinit;
      #endif
      ...

We recommend #ifdef protecting the dist pieces

Should be first thing in main "client" processes will not go beyond this line (they "live" in the constructor). Must persist throughout main

The template argument is the context type. It provides the compiler/runtime with all necessary type and graph information.

cholesky_dist_ready/*.h
Dist-ready 3

1. In cholesky.h
...
#ifdef _DIST_
    void serialize( CnC::serializer & ser )
    {
        // serialize members which are not collections
        ser & * & *;
    }
#endif

2. In cholesky_types.h
...
#ifdef _DIST_
CNC_BITWISE_SERIALIZABLE( triple );
#endif
...

Our context has 2 members "b" and "p" which need to be marshalled
Launching distributed CnC (sockets)

Clients and host: env DIST_CNC=SOCKETS

On Host, set CNC_SOCKET_HOST

1. number_of_clients
2. name_of_script

1. Host prints contact string to manually start clients
   CNC_SOCKET_CLIENT=<contact_string>

2. Host launches given script twice:
   1. -n must return number of clients
   2. Starting clients with given contact string (e.g. through ssh)

Example scripts for Windows and Linux are provided
Launching distributed CnC (MPI)

Clients and host: env DIST_CNC=mpi

[] env DIST_CNC=mpi mpirun -n <N> ./distcholesky 1000 100
Efficiency on distributed memory

- By default, work is distributed in round-robin
- Data follows upon request (bcast)
- It works, but often not very efficiently
DistCnC-ready performance (UTS)

- Unbalanced tree search
  - Tree shape unknown in advance
- CnC code is trivial
  - CnC: 151 loc
  - Shmem: ~1000 loc
  - MPI: ~800 loc
- CnC performs better on single node (multi-threaded)
- Performance gap in the mid-sized region is a load-balancing issue
  - Experimental version solves it
Default round-robin distribution can be arbitrarily bad.
Tuned Performance

CnC Time (tuned) [IB]

- inverse
- primes
- mandelbrot
- cholesky

Time [sec]

#nodes (24h-threads each)

2 4 8 16 32
Scalability Comparison

**Cholesky - Speedup**
(matrix 16kx16k, blocks 100x100)

**MatrixInverse - Speedup**
(matrix: 16kx16k, blocks: 90x90)
(Almost) seamless scalability in the cloud

- The unified programming model promises high scalability
- No fiddling with MPI or alike during development
- Services offering clusters with high-speed connectivity are coming up in the cloud
- CnC & cloud provide a convenient combination to scale from a single machine to a cluster
  - CnC eases the development (nothing extra to get a distributable SW)
  - Cloud services free you from maintaining complicated and costly HW
What we’ve learned

• Making a CnC application ready for distributed memory is a relatively small step
• No unnecessary barriers/syncs make it scale from 1 to thousands of cores
• CnC is a unified programming model for shared and distributed memory
Tuning

- CnC tuning influences how the runtime manages the interaction between CnC entities (items, steps, tags)
  - This is orthogonal to optimizations of the serial language (C++)
- The tuning expert gives hints to the runtime
  - For a specific application
- Examples
  - Garbage collection
    - Detect that an item will not be used any more
    - A semantic attribute, different from traditional GC
  - Pre-declaring dependencies
  - Influence step scheduling
  - Memoizing step execution
  - Work/data distribution
- Much of the tuning potential yet to come
Tuning Independently of Step Code

```cpp
class my_context
{
    ...
    CnC::tag_collection< int, CnC::preserve_tuner > tcoll;
    CnC::step_collection< my_step, my_tuner > scoll;
    ...

def my_context()
    : scoll( *this, stuner_instance ),
    tcoll( *this, ttuner_instance )
    ...
};

class my_tuner : public CnC::step_tuner
{
    int compute_on( int tag, my_context & ctxt ) const
    {
        return tag % numProcs();
    }
    ...
};
```

Tuner-types are optional arguments to each collection. Can be customized.

Custom tuners can be parametrizable at runtime.

Custom tuners implement a given interface (partially).
Garbage Collection

• Data-items can become “dead” if there will be no more gets to it
• Dead items can be removed and memory freed
• Current CnC cannot detect when an item becomes dead

• Programmer can provide the number of expected gets when an item is put
  – Semantic attribute
  – Providing get-counts is not always possible
  – It breaks if steps are fully executed more than once.
Writing a tuner (get_count)

- Write a class which implements (parts of) tuning interface
  - Derive from respective default to get default implementations
- Get_count is an attribute of data-items -> need a item-tuner (hashmap_tuner)
- Get_count function should return number of gets for given item (or NO_GETCOUNT)

```cpp
struct cholesky_tuner : public CnC::hashmap_tuner{
    ...
    int get_count( const triple & 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;
    }
    ...
};
```

No side-effects
Using the tuner

- Optional argument to collection type
- Optional argument to collection constructor
  - Needed only for parametrizable tuners

```cpp
// The context class
struct cholesky_context : public CnC::context< cholesky_context >
{
    ...
    // Item collections
    CnC::item_collection< triple, tile_type, ********* > tiles;
    ...
    // init tuners
    tuner( _p ),
    ...
    tiles( *this, "tiles", ***** ),
    ...
#define _DIST_
    void serialize( CnC::serializer & ser )
    {
        ser & tuner & p & b;
    }
#undef _DIST_
};
```

Provide tuner-type

Init tuner

Give it to item-collection

Our tuner needs marshalling

cholesky_getcount/cholesky.h
Get-count: compile and run

[] make cholesky
[] ./cholesky 1000 100

To show the effect
[] make cholesky OTPS=-g
[] gdb --args ./cholesky 1000 100
[gdb] break cholesky_cnc.cpp:184
[gdb] run
[gdb] ^z
[] ps
[] pmap –x <pid>

On windows
Breakpoint at cholesky_cnc.cpp:184
Run
Compare mem-usage in task-manager

Compare the summary with basic version
E.g. RSS is ~ 22100kb < 37300kb
What we’ve learned

• Tuning interface is separate from step code
• Tuners can be parametrizable
• Get-counts can reduce memory footprint
  – They are about application semantics
Reducing the number of re-schedules

• If the data dependences are known early re-schedules become unnecessary
• Tuner can provide data dependences
  – Runtime will not schedule steps until they are satisfied
Depends

• Data dependences are an attribute of steps -> need a step-tuner (step_tuner<>)
• Let’s use one tuner for items and steps

```cpp
struct cholesky_tuner : public CnC::step_tuner<>,
                    public CnC::hashmap_tuner
{
    ...

    template< typename T >
    void depends( const int tag, cholesky_context & c, T & dC ) const;

    template< typename T >
    void depends( const pair & tag, cholesky_context & c, T & dC ) const;

    template< typename T >
    void depends( const triple & tag, cholesky_context & c, T & dC ) const;
    ...
};
```

One for each step/tag-type

Opaque template argument allows compiler optimization

cholesky_getcount_deeps/cholesky.h
Assign tuner to step-collections

• Similar as for item-collections

```cpp
// The context class
struct cholesky_context : public CnC::context< cholesky_context >
{
    ...

    // Step Collections
    CnC::step_collection< cholesky_step, cholesky_tuner > cholesky;
    CnC::step_collection< trisolve_step, cholesky_tuner > trisolve;
    CnC::step_collection< update_step, cholesky_tuner > update;
    ...

    // init step colls
    cholesky( *this, tuner, "Cholesky" ),
    trisolve( *this, tuner, "Trisolve" ),
    update( *this, tuner, "Update" ),
    ...
};
```

**Provide tuner-type**

**Give the tuner to step-collections**

cholesky_getcount_depends/cholesky.h
Implementing depends

- For each to-be-consumed data item issue
dc.depends( <collection>, <tag> );

```cpp
template< typename T >
void cholesky_tuner::depends( const int tag, cholesky_context & c, T & dC ) const
{
    dC.depends( c.tiles, triple(tag, tag, tag) );
}
...

template< typename T >
void cholesky_tuner::depends( const triple & tag, cholesky_context & c, T & dC ) const
{
    const int k = tag[0], j = tag[1], i = tag[2];
    dC.depends( c.tiles, tag );
    if( i==j ) { // Diagonal tile.
        dC.depends( c.tiles, triple(k+1,j,k) );
    } else { // Non-diagonal tile.
        dC.depends( c.tiles, *********** );
        dC.depends( c.tiles, *********** );
    }
}
```

Declare the 2 dependences for update

cholesky_getcount Depends/cholesky.h
Depends: Effect

[] make cholesky
[] ./cholesky 1000 100

-> no more re-schedules

[]> ./cholesky 1000 100
The time taken for parallel execution a matrix of size 1000 x 1000 :
0.0704816 sec
Steps created( 220 )
Steps scheduled( 220 ) inflight( 0, 0 )
What we’ve learned

- Depends can reduce number re-schedules
  - It’s about application semantics
- We can share one tuner among collection(-type)s
More Tuning Available

• Tag-ranges (configurable partitioner, range-sizes)
• Storage types (hash-map, vector)
• Tag memoization
• Priorities
• Pre-scheduling
• Expected in 0.9: cancelation

• Work and data distribution (distributed memory)
CnC makes work and data distribution easy and efficient

- By default, a simple local round-robin scheduling is done
  - Data is sent to where needed (requested)

- Tuner can declare
  - where work is to be executed:
    ```cpp
    int tuner::compute_on( step_tag )
    ```
  - where the data needs to go:
    ```cpp
    int tuner::consumed_on( item_tag ) or
    vector< int > tuner::consumed_on( item_tag )
    ```

- Both returning ranks/ids of address-spaces
- Mapping tag->rank can be computed
  - Statically at compile time
  - Dynamically at init-time
  - Dynamically on the fly
Increased Productivity when experimenting with distribution plans

- With the producer/consumer edges CnC can couple data and work
- Allows creating a distribution plan at a single place
  - For each item, declare steps which depend on it ("consumed_by")
  - With compute_on the runtime knows where to send the item to
  - Runtime auto-distributes data optimally according to declared work-distribution
  - Depends on the program semantics only

```cpp
int tuner::consumed_on( const tag t ) const
{
    return compute_on( consumed_by( t ) );
}
```
Changing distribution (cholesky)

Optimal distribution might depend on several factors.

CnC makes it easy to customize.

Diagram: Cholesky - Speedup
(matrix 16kx16k, blocks 100x100)

- Speedup over single-node MKL-LAPACK
- #nodes (24 h-threads each)

Legend:
- MKL-LAPACK
- CnC COLUMN_CYCLIC
- CnC BLOCKED_ROWS
- CnC ROW_CYCLIC
- CnC BLOCKED_CYCLIC
- MKL-scaLAPACK
A flexible distribution plan

• Provided skeleton implements consumed_by mechanism
  – Cholesky semantics a bit too complex for this tutorial
  – The code is extremely small compared to what would be needed in a message passing framework (40 lines)
  – It’s separate from the step-code!
  – See next slide if you’re interested

• consumed_on and 3x compute_on all go through a compute_on function

• Let’s experiment with that one to change the distribution plan
// 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 triple & 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 ) ); // cholesky 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 ) ); // trisolve only

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

  if( _i == _k-1 ) {
    if( _i == _j ) { // on diagonal on trisolve
      for( int j = _k; j < m_p; ++j ) {
        mark( compute_on( pair( _k - 1, j ), m_context ), _d );
      }
    } else { // update
      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;
}
Row-cyclic and Column cyclic

```c
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 ********************;
        break;
    case COLUMN_CYCLIC :
        return ********************;
        break;
    case BLOCKED_CYCLIC :
        return ( (i/2) * n + (j/2) ) % numProcs();
        break;
    }
}
```

Fill in what’s needed for row-cyclic and column-cyclic

`cholesky_getcount_depends_dist/cholesky.h`
Distribution plan

[] make distcholesky
[] env DIST_CNC=MPI mpirun -n 4 ./distcholesky 1000 100

• Let’s confirm that it really changes the distribution
  – Return a constant and watch with top
  – Return i % 2 and watch with top
What we’ve learned

• Much of the tuning can be done on the semantic level
• Defining a distribution is fully separate from the domain code
• Only little additional code needed to achieve an efficient distribution
Pointers are friends

• Pointers can minimize memory pressure
• What about dynamic single assignment (DSA)?
  – The use of pointers is orthogonal to obeying DSA
  – Your items can well be pointers **as long as their content is not altered after putting**
• But then what about distributed memory?
  – distCnC supports pointers
    – The content must be serializable
    – Use CNC_POINTER_SERIALIZABLE to declare your pointers
      (again, safety first)
• Super, but memory deallocation is a problem!?!?
  – You can use get-count garbage collection by just using smart-pointers
    – std::shared_ptr is supported out of the box
Cholesky and smart pointers

• Only changes are
  – New typedefs
    ```cpp
typedef std::shared_ptr< const tile_type > tile_const_ptr_type;
typedef std::shared_ptr< tile_type > tile_ptr_type;
```

  – Using those as the item-type
    ```cpp
    // Item collections
    CnC::item_collection< triple, tile_const_ptr_type, cholesky_tuner > tiles;
    ```

  – When accessing Tiles, we now need to dereference
    ```cpp
    (*A_block)( k_b, k_b );
    ```

• That’s it!

Using const-pointers makes it more likely that the compiler finds DSA violations.
What we’ve learned

• Pointers are friends
(Almost) seamless scalability in the cloud

- The unified programming model promises high scalability
- No fiddling with MPI or alike during development
- Services offering clusters with high-speed connectivity are coming up in the cloud
- CnC & cloud provide a convenient combination to scale from a single machine to a cluster
  - CnC eases the development (nothing extra to get a distributable SW)
  - Cloud services free you from maintaining complicated and costly HW
The cloud and CnC in action

• In dir ~/cloud there are 2 (simple) scripts
  – mk_pbdc.sh: Layout and startup a cluster in the cloud
  – run_dcnc_pb.sh: Launch your (local) CnC application on that cluster

• The cluster is up and running
  – To see what it provides, do a ./mk_pbdc.sh -h
  – You need “cluster-ips” to start jobs

• To start a job (must be in ~/cloud), do

```bash
[] ./run_dcnc_pb.sh -h cluster.ips -p <path-to-cholesky> \
  -a 2000 100
```

• Try other apps from the distro
  – Black-scholes
  – RTM
  – Primes
  – Mandelbrot
Backup
Intel® Concurrent Collections for C++

Three Separate Ingredients of a CnC application

- **Domain (CnC API)**
  - Defines structure and semantics of the program
  - No explicit parallelism

- **CnC Runtime**
  - Coordinates & parallelizes
  - Hides the complexity of parallelization

- **Tuning (CnC API)**
  - Controls the runtime and application; is specific to platform
  - Not intertwined with domain

**Compiler C++**

**CnC Application**

**Native binary**
Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2®, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Legal Disclaimer

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

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


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

Copyright © 2012. Intel Corporation.

http://intel.com/software/products