Ct: C for Throughput Computing

Channeling NeSL and SISAL through C++

Mohan Rajagopalan
Anwar Ghuloum

Looking Backwards and Forwards

- "There are no new ideas..."
  - Silicon trends introduce new opportunities to revisit
    - Parallel programming models
    - Parallel applications/algorithms
  - …on a much different scale
- "…but much room for improvement…"
  - Modern programming methods require rethinking
    - Dynamic compilation, managed runtimes
    - Fine grained modularity
    - Exceptionally complex and diverse patterns in single applications
      - Cf: Games!
- "…and new usages."
  - Parallel incremental/adaptive (re)computation
  - Forward scaling

What Software Vendors are Telling Us

- Strong interest by ISVs for a parallel programming model which is:
  - **Easy to use *and* high performance: sounds difficult already!**
  - **Portable**: Desire the flexibility to target various HW platforms and adapt to future variations
- Programming parallel applications is 10,100,1000x* *less* productive than sequential
  - Non-deterministic programming errors
  - Performance tuning is extremely microarchitecture-dependent
- Parallel HW is here today, better programming tools are needed to take advantage of these capabilities
  - Quad core on desktop arrived nearly a year months ago
  - Multi- and Many-core DP and MP machines are on the way
  - (Also, programmable GPUs going on 8 years)

**Depends on which developer you ask.*

Why We Started With Ct

C with Explicit Parallelism alone:

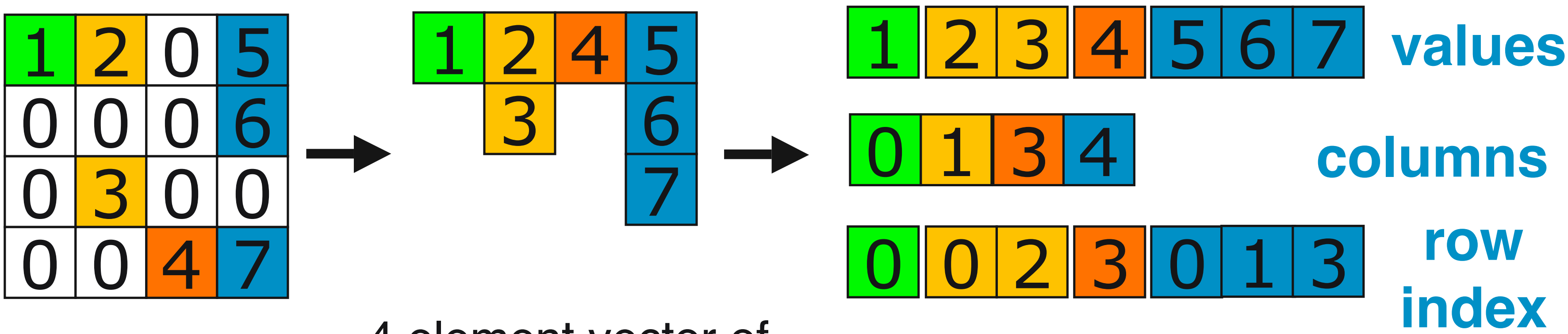
172 lines of code

- We moved from video algorithms to physics kernels
  - Rigid Body Dynamics
    - Broad and narrow-phase collision
    - Solvers
  - Cloth Simulation
- Found it painful to program using "legacy" parallel programming models
- Not surprisingly, same concerns as software vendors
- (Nested) data parallel models make it easier

Ct: <6 lines of code, faster, scalable

```
TVEC<F64> smvpCSC(TVEC<F64> A,
          TVEC<I32> cols, TVEC<F64> v)
{
    TVEC<F64> expv, product, result;
    expv =  distribute(v,cols);
    product = A*expv;
    return addReduce(product);
}
```

Irregular Data Structures



4x4 sparse matrix

4 element vector of variable length vectors

Flattened representation with column & row metadata

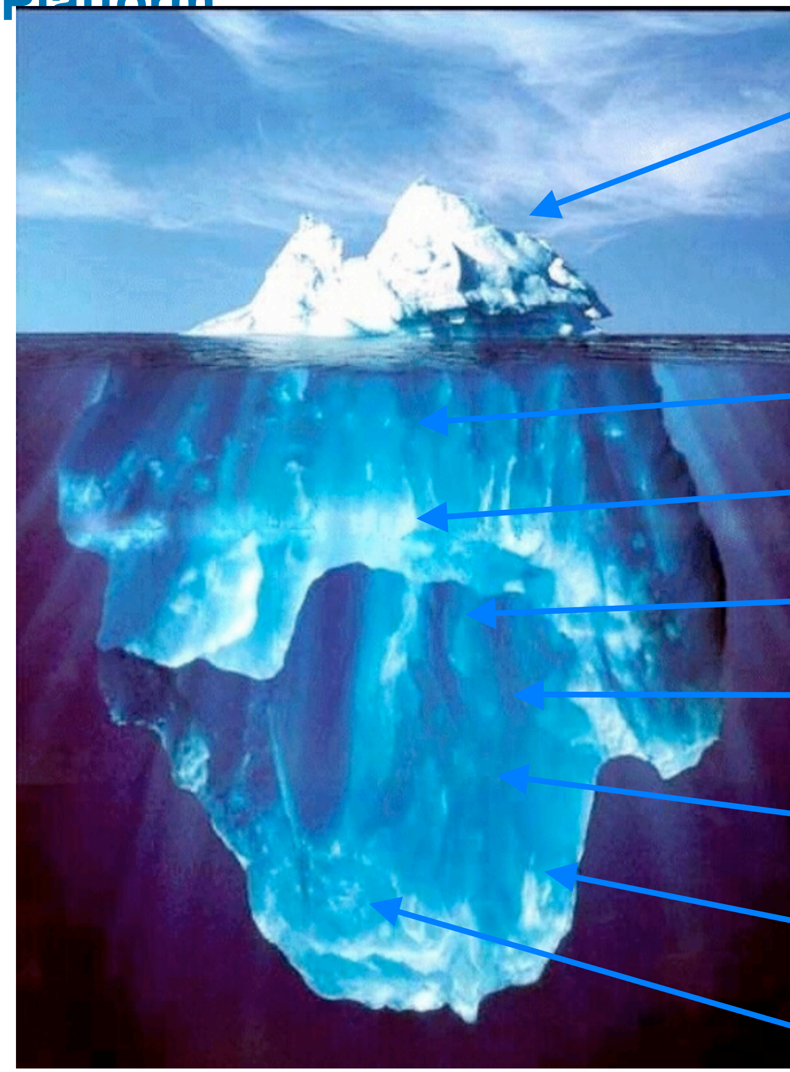
- A classic example: Sparse matrices
  - Common in RMS applications
  - Difficult for a programmer to deal with

Nested data parallelism handles irregular structures automatically

Why Dataflow is Interesting

- Data isolation
  - Spatio-temporal localization of effects leads to desirable properties for parallelize
    - →Locality preserved
    - →Safety is guaranteed
- Required agility for many-core
  - *Scaling*
    - >Stretching "horizontally" to more threads, smaller footprints
    - >Stretching "vertically" to control memory bandwidth, arithmetic intensity
  - *Adaptivity*
    - >For incremental recomputation
    - >Intelligent, scalable synchronization/scheduling algorithms

Language Vehicle for General Purpose Parallel Programming Platform



Ct Api
- Nested Data Parallelism
- *Deterministic Task Parallelism*

Deterministic parallel programming

Fine grained concurrency and synch

Dynamic (JIT) compilation

High-performance memory management

Forward-scaling binaries for SSEx, ISAx

Parallel application library development

Performance tools for future architectures

What Is Ct?

"Extending" C++ for Throughput-Oriented Computing

- Ct adds new data types (parallel vectors) & operators to C++
  - Library interface and is ANSI/ISO-compliant
- Ct abstracts away architectural details
  - Vector ISA width / Core count / Memory model / Cache sizes
- Ct forward-scales software written today
  - Ct platform-level API is designed to be *dynamically* retargetable to SSE, SSEx, ISA x, etc
- Ct is deterministic*
  - No data races

Nested data parallelism and *deterministic task parallelism* differentiate Ct on parallelizing irregular data and algorithm

The Ct Surface API: Nested Data Parallelism ++

TVECs

The basic type in Ct is a TVEC

- TVECs are managed by the Ct runtime
- TVECs are single-assignment vectors
- TVECs are (opaquely) flat, multidimensional, sparse, or nested
- TVEC values are created & manipulated exclusively through Ct API

Declared TVECs are simply references to immutable values

```
TVEC<F64> DoubleVec;   // DoubleVec can refer to any vector of doubles
…
DoubleVec = Src1 + Src2;
…
DoubleVec = Src3 * Src4;
```

Assigning a value to DoubleVec doesn't modify the value representing the result of the add, it simply refers to a *new* value.

Ct In Action: C User Migration Path using Vector-style

```
float s[N], x[N], r[N], v[N], t[N];
float result[N];

for(int i = 0; i < N; i++) {
  float d1 = s[i] / ln(x[i]);
  d1 += (r[i] + v[i] * v[i] * 0.5f) * t[i];
  d1 /= sqrt(t[i]);
  float d2 = d1 – sqrt(t[i]);

  result[i] = x[i] * exp(r[i] * t[i]) *
    ( 1.0f - CND(d2)) + (-s[i]) * (1.0f - CND(d1));
}
```

```
#include <ct.h>

T s[N], x[N], r[N], v[N], t[N];
T result[N];
TVEC<T> S(s, N), X(x, N), R(r, N), V(v, N), T(t, N);

TVEC<T> d1 = S / ln(X);
d1 += (R + V * V * 0.5f) * T;
d1 /= sqrt(T);
TVEC<T> d2 = d1 – sqrt(T);

TVEC<T> tmp = X * exp(R * T) *
   ( 1.0f - CND(d2)) + (-S) * (1.0f - CND(d1));

tmp.copyOut(result, N);
```

(Callouts: 1 – `#include <ct.h>`; 2 – TVEC declarations; 3 – `for` loop; 4 – loop body; 5 – `tmp.copyOut(result, N);`)

Use Animation

Ct in Action: Kernel-style Programming with Ct Lambdas

```
TElt2D<F16> threebythreefun(TElt2D<F16> arg, F32 w0, F32 w1, F32 w2,
                                                   F32 w3, F32 w4) {
 return w0*arg +
         w1*arg[-1][0] +
         w2*arg[0][-1] +
         w3*arg[1][0] +
         w4*arg[0][1];
};
```

Ct element-wise function

Element-wise argument and result

Relative indexing for neighboring values

```
TElt2D<I8> errordiffuse(TElt2D<F16> pixel) {
   return someexpression(pixel,RESULT[-1][0],RESULT[-1][-1],RESULT[0][-1]);
};

TVEC2D<F16, defaultvalue> colorplane, filteredcolors;
TVEC2D<I8, defaultvalue> ditheredcolors;
…
filteredcolors = map(threebythreefun, arg, 1/2, 1/8, 1/8, 1/8, 1/8);
ditheredcolors = map(errordiffuse, filteredcolors);
```

Dependences on neighboring results (wavefront pattern)

Simple interface for applying these "kernels"

The Ct Threading Model

Dataflow is back!

One way of looking at Ct:

A declarative way to specify complex task graphs

What we needed:

- Fine-grained concurrency and synchronization support
  - A bunch of lightweight tasks arranged in a dependency graph
- Novel optimizations and usage patterns
  - Reuse of task graph (called *future-graph*)
  - Incremental/adaptive update of FG

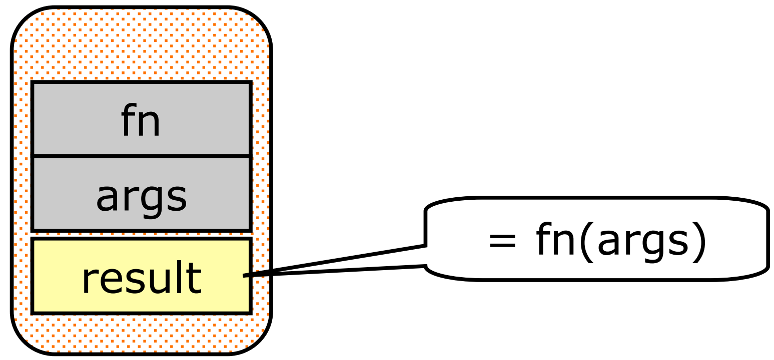
What we came up with:

- A super-lightweight *futures*-based threading abstraction
- Primitives for bulk creation of futures and complex synchronization

  → *Building blocks for dataflow-style task graphs*
- Composable first-class objects to enable dynamic optimization

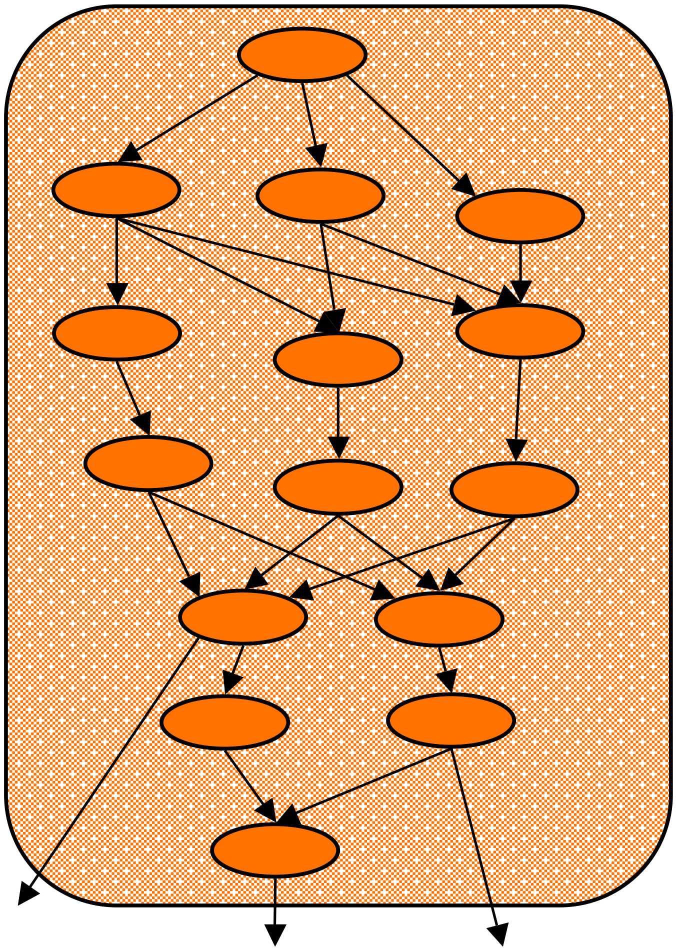
Feather-weight “Threads”: Futures

Futures: (Almost) stateless task



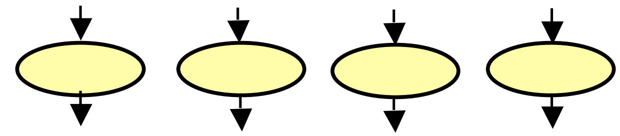
- API: Spawn & Read
- Futures can be in one of 3 states
  - Unevaluated: can be “stolen” or evaluated by reader
  - Evaluating: reader should wait for the result
  - Evaluated: reader can just grab the result
- Scheduled using distributed queues
  - Enqueued futures serviced by underlying worker threads
- Futures-creation about 2-3 orders of magnitude less expensive than thread creation

Simplifying Complexity through Data-parallel Patterns

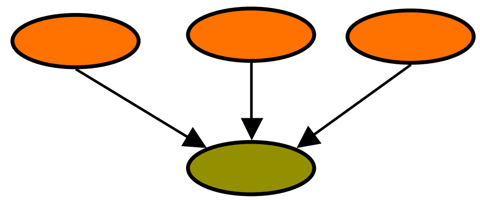


Element-wise operations

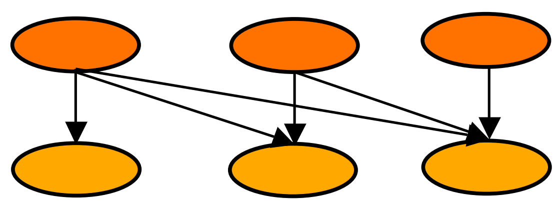
e.g. A[] = B[]+C[]



Reduction

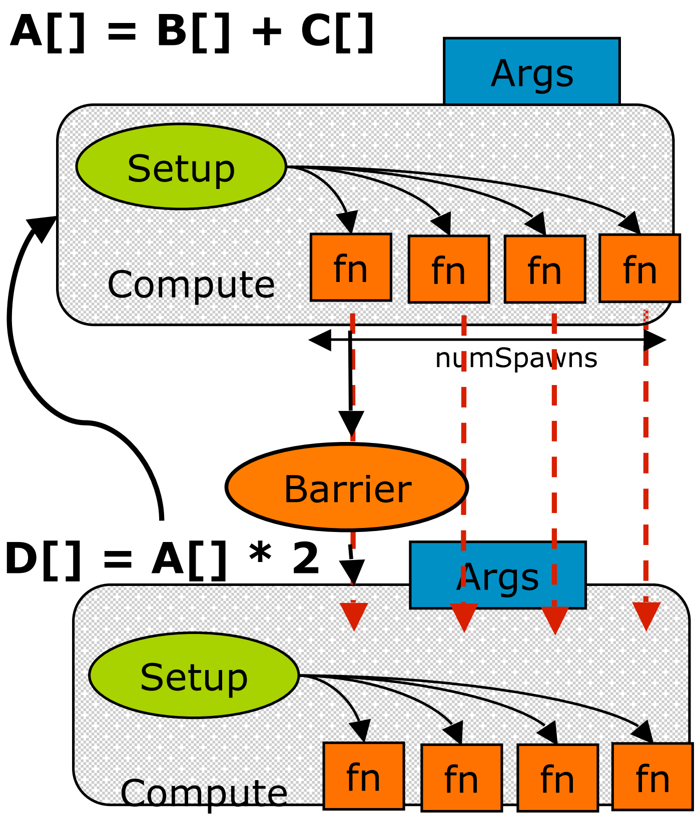


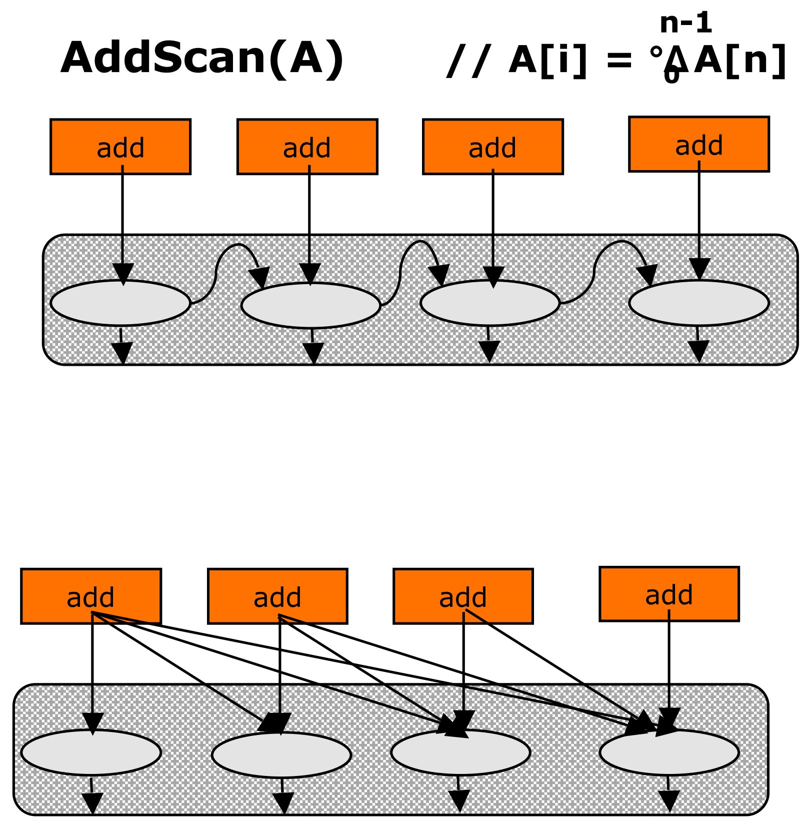
Prefix



High-Level Primitives

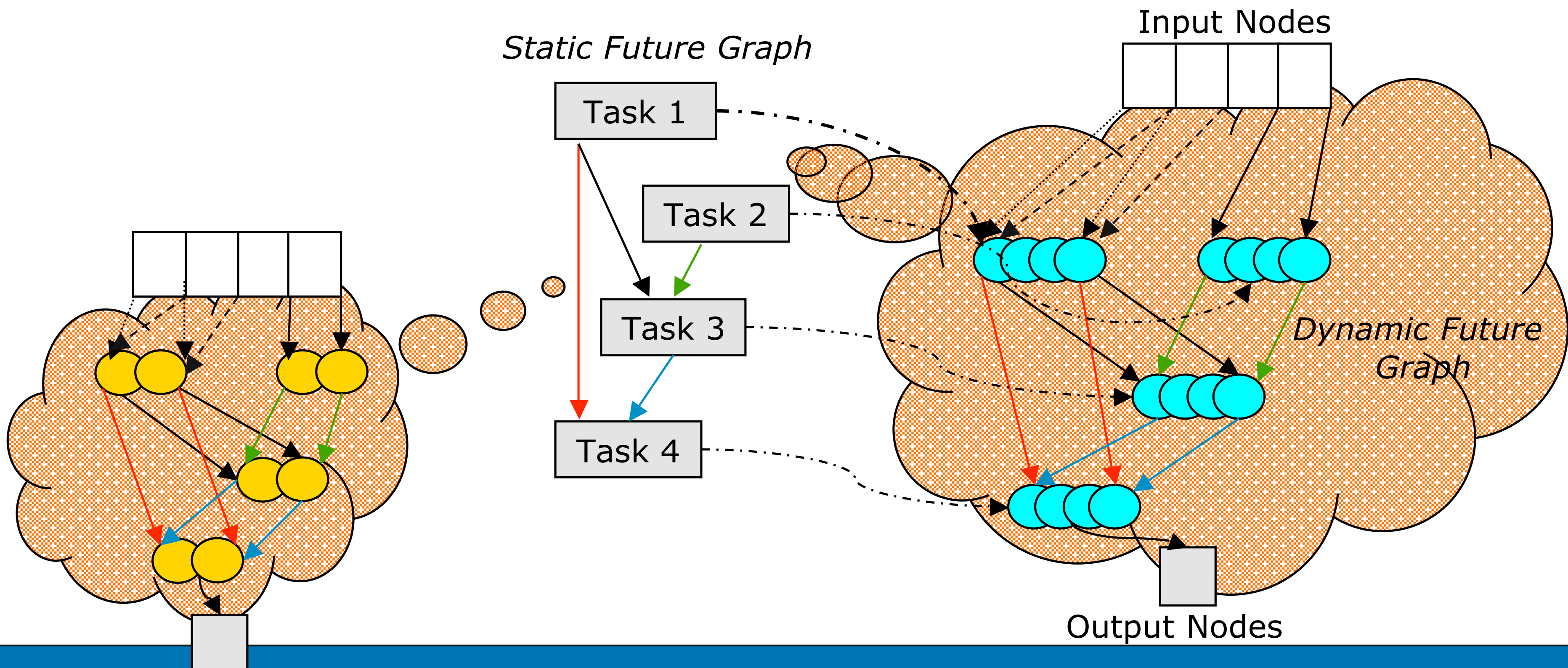
- Enable automatic dynamically configurable parallelism





Future Graphs Reuse and Adaptivity

- Abstraction for collectively manipulating about groups of futures
  - Generic reuse in code (esp. loops)
  - Play with funky scheduling algorithms
- 3 Basic operations: Creation, Instantiation, Evaluation



Task Parallelism in Ct

Two options:

Futures and HSTs (Hierarchical, Synchronous Tasks)

- Futures
  - Basically, any Ct Function/Lambda can be spawned off as an parallel task (can include both scalar and vector code)
- HSTs
  - A sensible generalization of Bulk Synchronous Processes
  - Regions can be hierarchical
  - Bodies of tasks can be mix of data parallel and scalar code

- More details: offline

What Is Ct?

- Ct adds new data types (parallel vectors) & operators to C++
  - Library interface and is ANSI/ISO-compliant
- Ct abstracts away architectural details
  - Vector ISA width / Core count / Memory model / Cache sizes
- Ct forward-scales software written today
  - Ct platform-level API is designed to be *dynamically* retargetable to SSE, SSEx, ISA x, etc
- Ct is deterministic*
  - No data races

Nested data parallelism and *deterministic task parallelism* differentiate Ct on parallelizing irregular data and algorithms

For more information: www.intel.com/go/Ct

intel®
Leap ahead™