INTEL® HPC DEVELOPER CONFERENCE

FUEL YOUR INSIGHT

INTEL® HPC DEVELOPER CONFERENCE
FUEL YOUR INSIGHT

USING C++ AND INTEL THREADING BUILDING BLOCKS TO PROGRAM ACROSS PROCESSORS AND CO-PROCESSORS

Evgeny Fiksman, Sergey Vinogradov and Michael Voss

Intel Corporation

November 2016

Intel® Threading Building Blocks (Intel® TBB)

Celebrating it's 10 year anniversary in 2016!

A widely used C++ template library for parallel programming

What

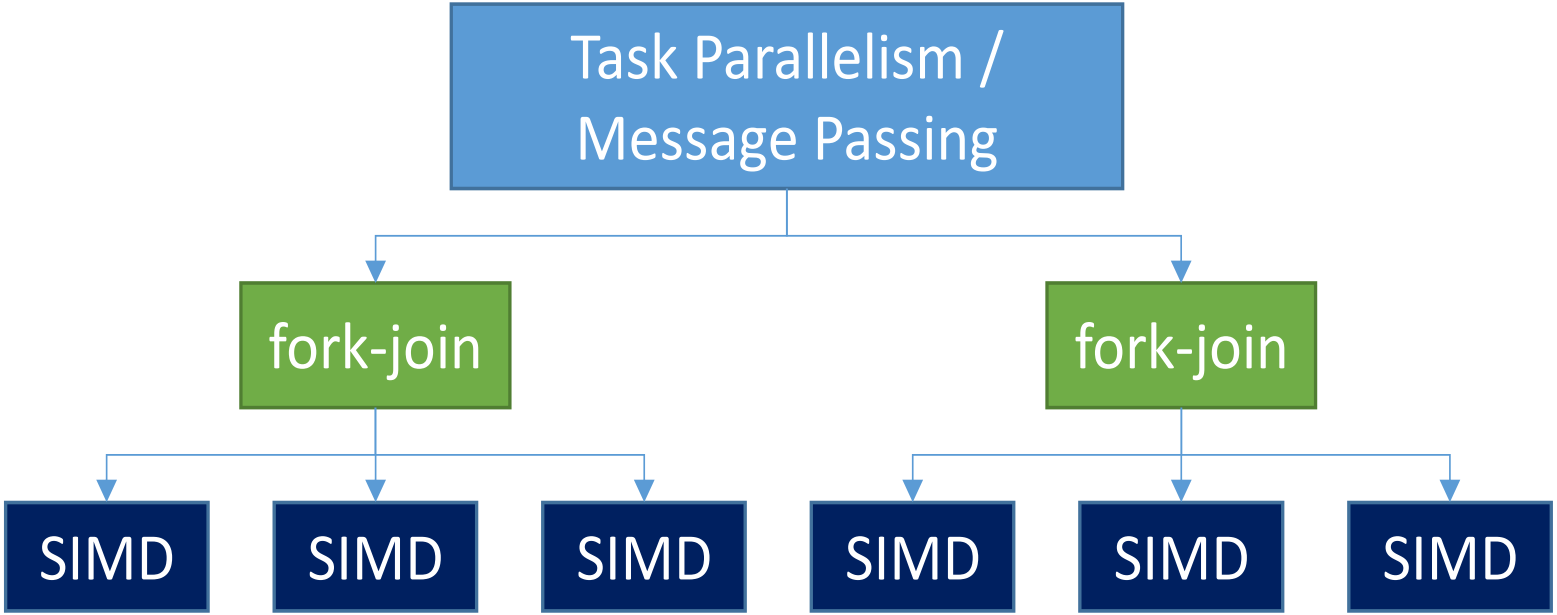
Parallel algorithms and data structures
Threads and synchronization primitives
Scalable memory allocation and task scheduling

Benefits

Is a library-only solution that does not depend on special compiler support
Is both a commercial product and an open-source project
Supports C++, Windows*, Linux*, OS X*, Android* and other OSes
Commercial support for Intel® Atom™, Core™, Xeon® processors and for Intel® Xeon Phi™ coprocessors

http://threadingbuildingblocks.org http://software.intel.com/intel-tbb

Applications often contain three levels of parallelism



Task Parallelism / Message Passing

fork-join

fork-join

SIMD

SIMD

SIMD

SIMD

SIMD

SIMD

Intel® Threading Building Blocks

threadingbuildingblocks.org

- 🟦 Parallel algorithms and data structures
- 🟩 Threads and synchronization
- 🟧 Memory allocation and task scheduling

Generic Parallel Algorithms

Efficient scalable way to exploit the power of multi-core without having to start from scratch.

Flow Graph

A set of classes to express parallelism as a graph of compute dependencies and/or data flow

Concurrent Containers

Concurrent access, and a scalable alternative to serial containers with external locking

Synchronization Primitives

Atomic operations, a variety of mutexes with different properties, condition variables

Task Scheduler

Sophisticated work scheduling engine that empowers parallel algorithms and flow graph

Thread Local Storage

Unlimited number of thread-local variables

Threads

OS API wrappers

Miscellaneous

Thread-safe timers and exception classes

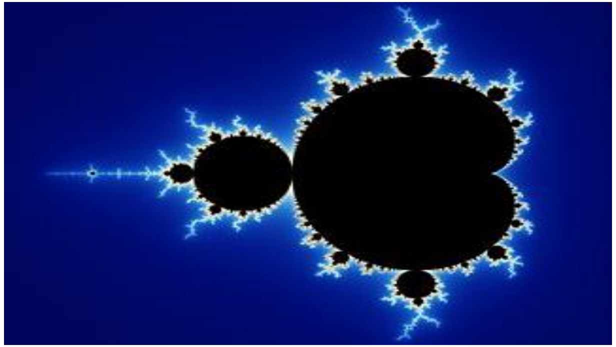
Memory Allocation

Scalable memory manager and false-sharing free allocators

Mandelbrot Speedup

Intel® Threading Building Blocks (Intel® TBB)

```
int mandel(Complex c, int max_count) {
  int count = 0; Complex z = 0;
  for (int i = 0; i < max_count; i++) {
    if (abs(z) >= 2.0) break;
    z = z*z + c; count++;
  }
  return count;
}
```



Task is a function object

Parallel algorithm

```
parallel_for( 0, max_row,
  [&](int i) {
    for (int j = 0; j < max_col; j++)
      p[i][j]=mandel(Complex(scale(i),scale(j)),depth);
  }
);
```

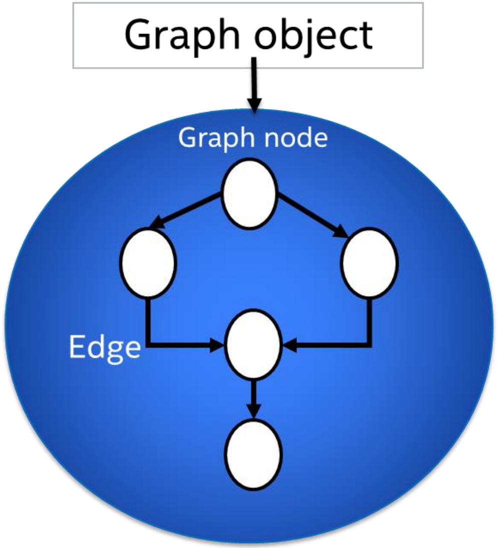
Use C++ lambda functions to define function object in-line

Intel Threading Building Blocks flow graph

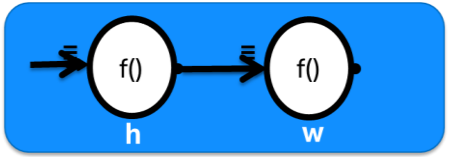
Efficient implementation of dependency graph and data flow algorithms

Design for shared memory application

Enables developers to exploit parallelism at higher levels

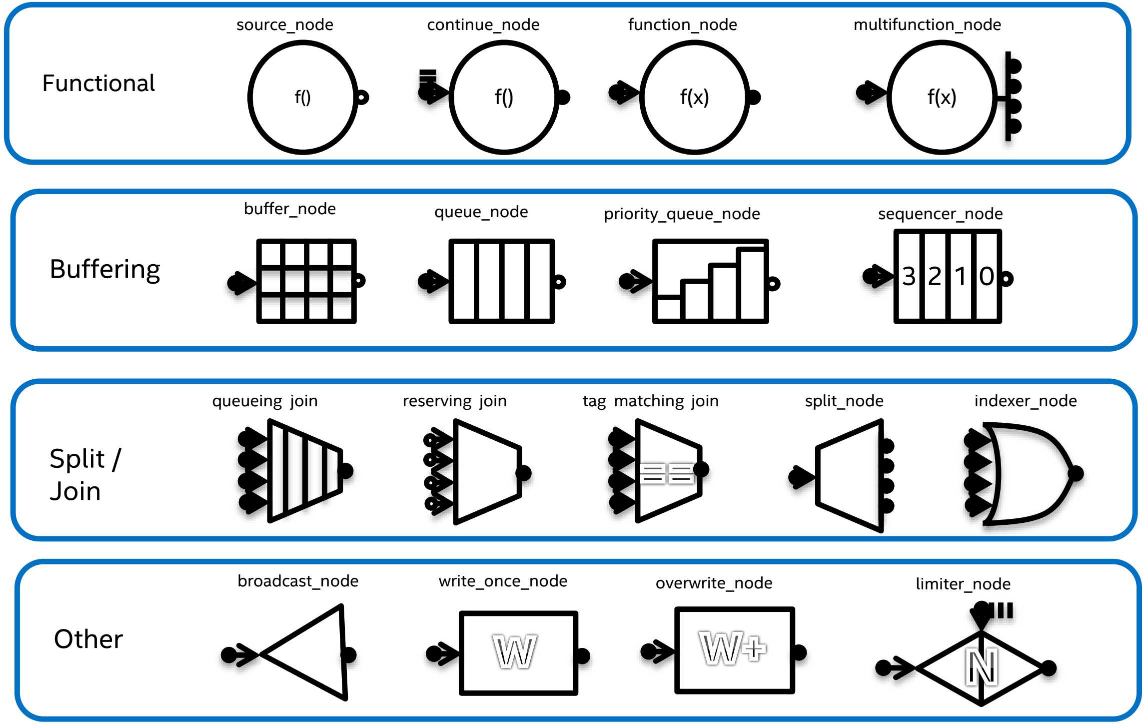


Hello World



```
graph g;
continue_node< continue_msg > h( g,
     []( const continue_msg & ) {
          cout << “Hello “;
     } );
continue_node< continue_msg > w( g,
     []( const continue_msg & ) {
          cout << “World\n“;
     } );
make_edge( h, w );
h.try_put(continue_msg());
g.wait_for_all();
```

Intel TBB Flow Graph node types:



Functional

source_node
continue_node
function_node
multifunction_node

Buffering

buffer_node
queue_node
priority_queue_node
sequencer_node

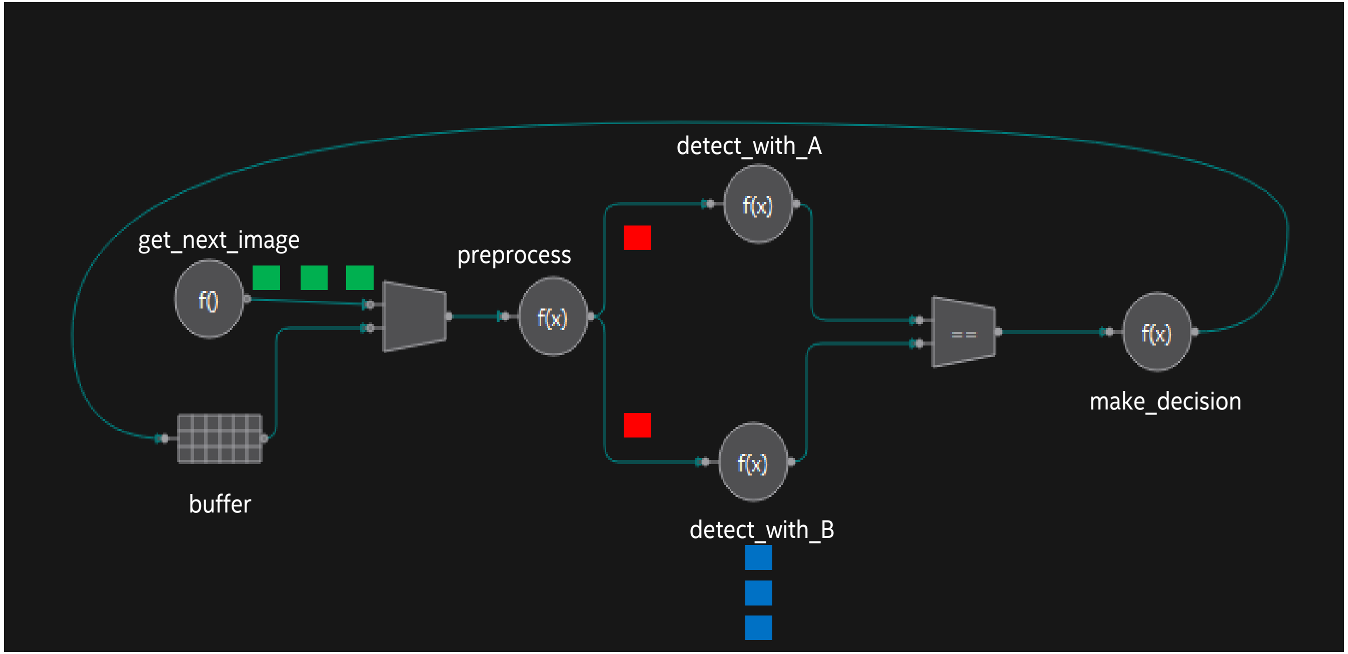
Split /
Join

queueing join
reserving join
tag matching join
split_node
indexer_node

Other

broadcast_node
write_once_node
overwrite_node
limiter_node

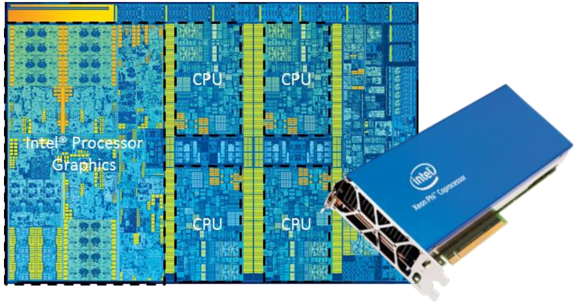
An example feature detection algorithm



Can express pipelining, task parallelism and data parallelism

Heterogeneous support in Intel® TBB

Intel TBB as a coordination layer for heterogeneity that provides flexibility, retains optimization opportunities and composes with existing models



FPGAs, integrated and discrete GPUs, co-processors, etc…

+

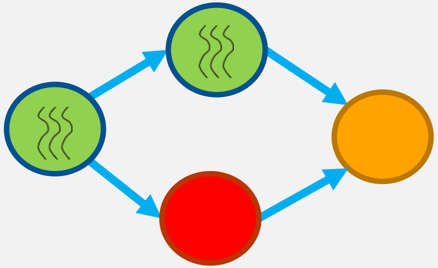
Intel® Threading Building Blocks
OpenVX*
OpenCL*
COI/SCIF
DirectCompute*
Vulkan*
….

Intel TBB as a **composability layer** for library implementations

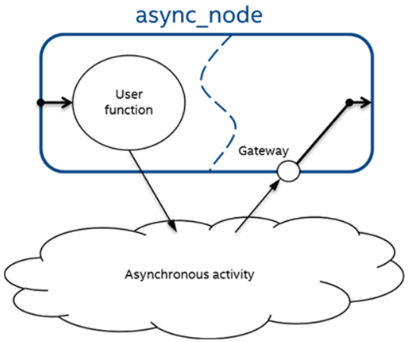
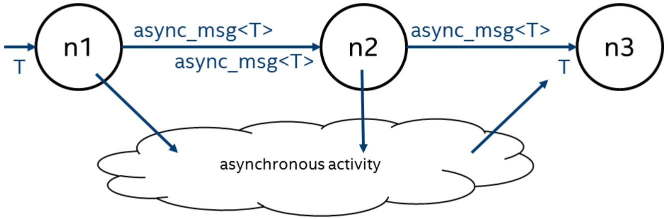
- One threading engine ***underneath*** all CPU-side work

Intel TBB flow graph as a **coordination layer**

- Be the glue that connects hetero HW and SW together
- Expose parallelism between blocks; simplify integration

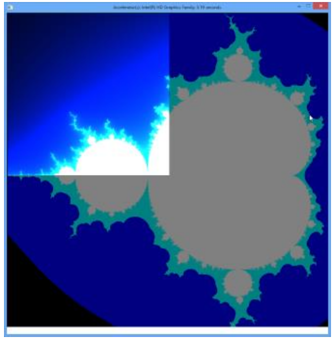


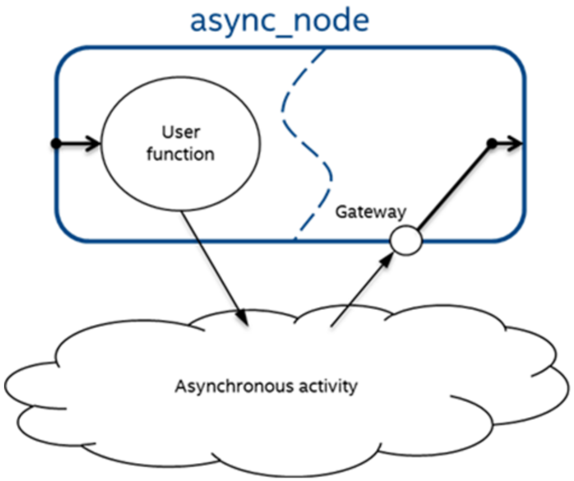
Support for Heterogeneous Programming in Intel TBB
So far all support is within the flow graph API

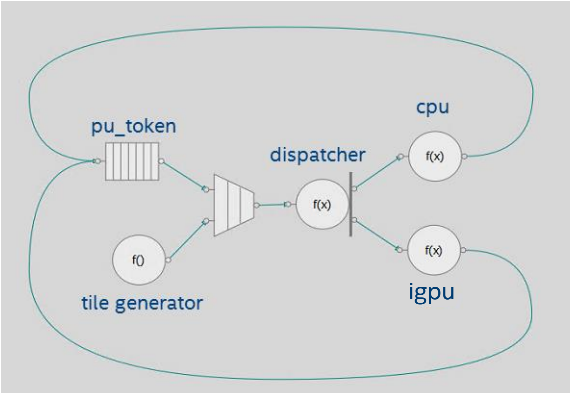
Feature	Description	Diagram
async_node<Input,Output	Basic building block. Enables async communication from a single/isolated node to an async activity. User responsible for managing communication. Graph runs on host.	
async_msg<T> *Available as preview feature*	Basic building block. Enables async communication with chaining across graph nodes. User responsible for managing communication. Graph runs on the host.	

async_node example

- Allows the data flow graph to offload data to any asynchronous activity and receive the data back to continue execution on the CPU

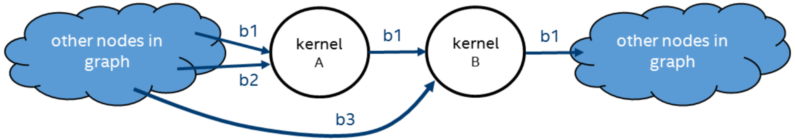
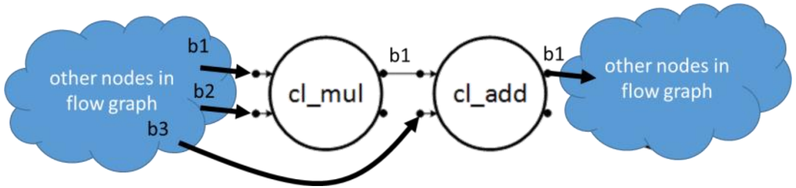




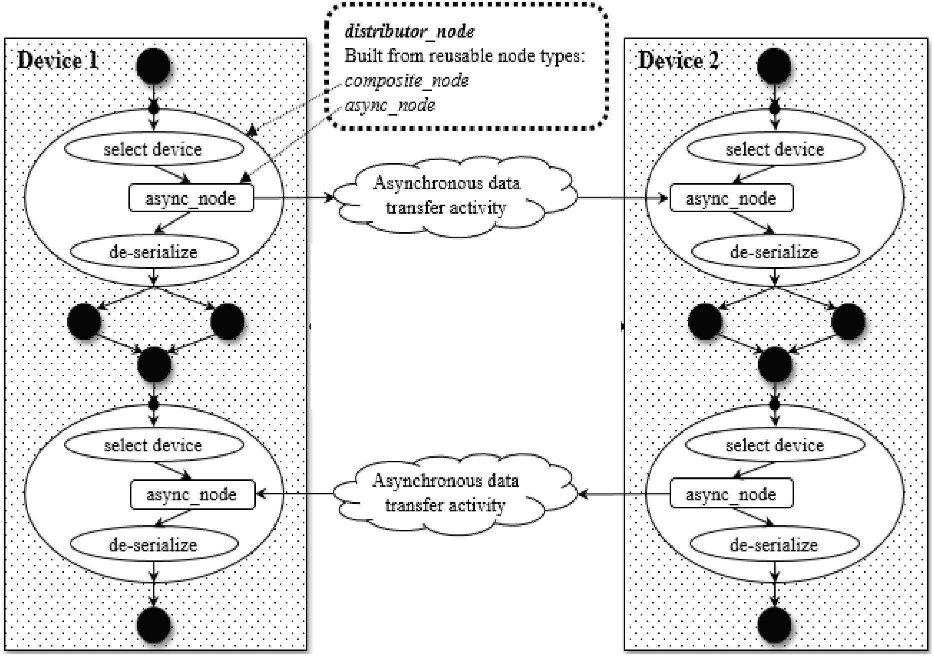


async_node makes coordinating with any model easier and efficient

Support for Heterogeneous Programming in Intel TBB
So far all support is within the flow graph API

Feature	Description	Diagram
streaming_node *Available as preview feature*	Higher level abstraction for streaming models; e.g. OpenCL, Direct X Compute, GFX, etc.... Users provide Factory that describes buffers, kernels, ranges, device selection, etc... Uses async_msg so supports chaining. Graph runs on the host.	
opencl_node *Available as preview feature*	A specialization of streaming_node for OpenCL. User provides OpenCL program and kernel and runtime handles initialization, buffer management, communications, etc.. Graph runs on host.	

Proof-of-concept: distributor_node



NOTE: async_node and composite_node are released features; distributor_node is a proof-of-concept

An example application: STAC-A2*

The STAC-A2 Benchmark suite is the industry standard for testing technology stacks used for compute-intensive analytic workloads involved in pricing and risk management.

STAC-A2 is a set of specifications

- For **Market-Risk Analysis**, proxy for real life risk analytic and computationally intensive workloads
- *Customers define the specifications*
- *Vendors implement the code*
- Intel first published the benchmark results in Supercomputing'12
  - http://www.stacresearch.com/SC12_submission_stac.pdf
  - http://sc12.supercomputing.org/schedule/event_detail.php?evid=wksp138

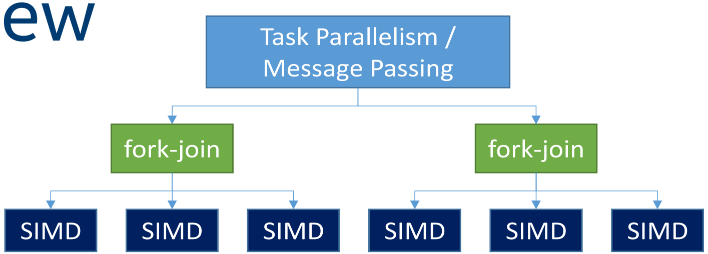
STAC-A2 evaluates the Greeks For American-style options

- Monte Carlo based Heston Model with Stochastic Volatility
- Greeks describe the sensitivity of price of options to changes in parameters of the underlying market
  - Compute 7 types of Greeks, ex: Theta – sensitivity to the passage of time, Rho – sensitivity for the interest rate

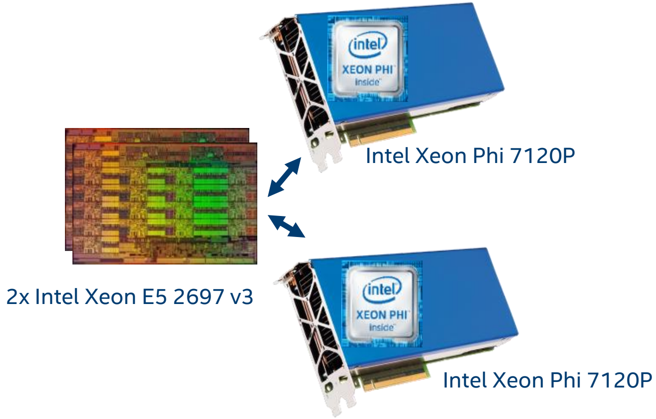
STAC™ BENCHMARK COUNCIL

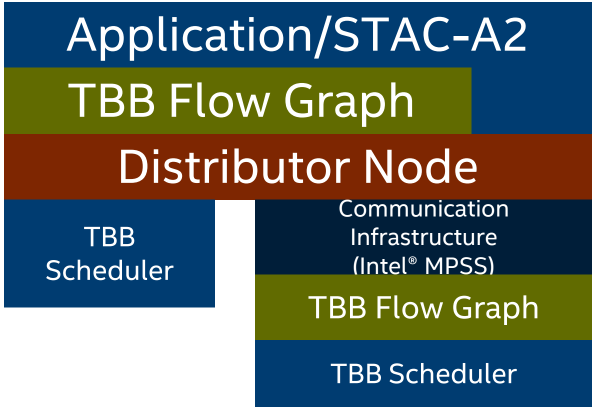
* "STAC" and all STAC names are trademarks or registered trademarks of the Securities Technology Analysis Center LLC.

STAC-A2 Implementation Overview

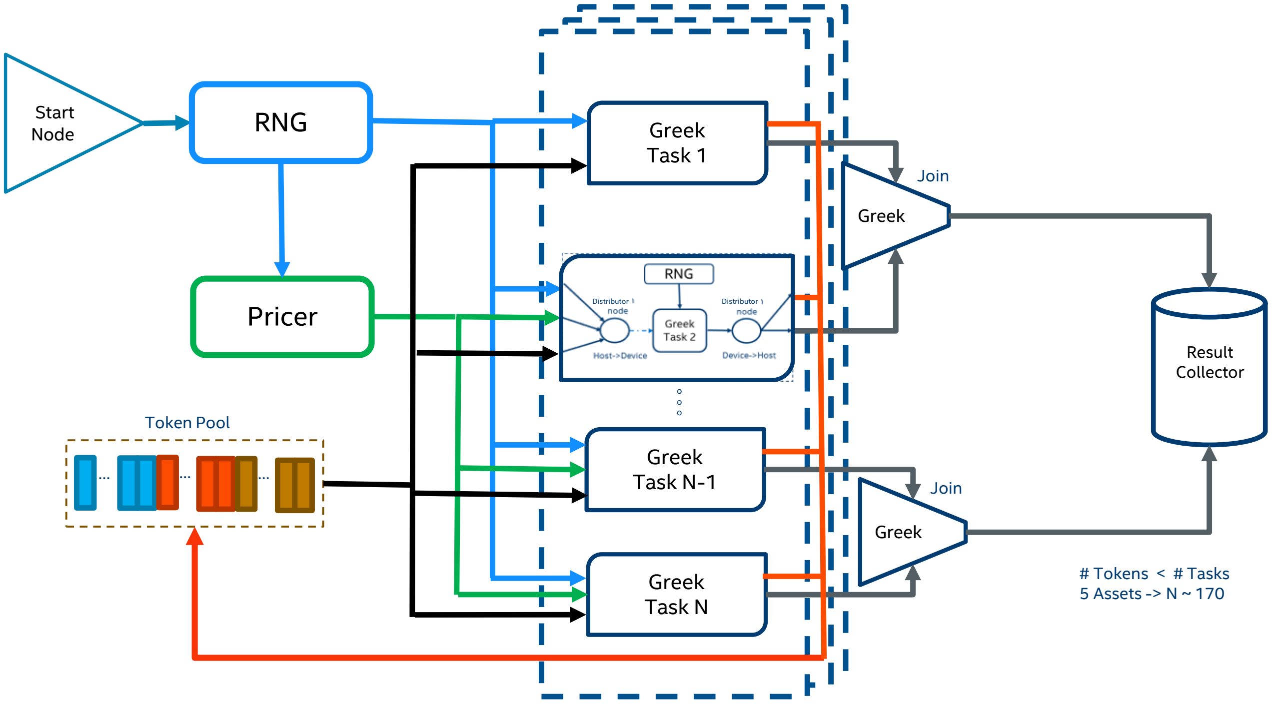


- Implemented with:
  - Intel TBB flow graph for task distribution
  - Intel TBB parallel algorithms for for-join constructs
  - Intel Compiler & OpenMP 4.0 for vectorization
  - Intel® Math Kernel Library (Intel® MKL) for RND generation and Matrix operations
- Uses asynchronous support in flow graph to implement “Distributor Node” and offload to the Intel Xeon Phi coprocessor - heterogeneity
- Using a token-based approach for dynamic load balancing between the main CPU and coprocessors





Intel TBB flow graph design of STAC-A2



Tokens < # Tasks
5 Assets -> N ~ 170

The Fork-Join & SIMD layers

Fork-Join , Composable with Flow Graph

SIMD Layer

```
for (unsigned int i = 0;  i < nPaths; ++i){

     double mV[nTimeSteps];
     double mY[nTimeSteps];
     ...
        for (unsigned int t = 0; t < nTimeSteps; ++t){


              double currState = mY[t] ;
              ....
              double logSpotPrice = func(currState, ...);
              mY[t+1] = logSpotPrice * A[t];
              mV[t+1] = logSpotPrice * B[t] + C[t] * mV[t];
              price[i][t] = logSpotPrice*D[t] +E[t] * mV[t];

        }

}
```

```
for (unsigned i = 0; i < nPaths; ++i)
{
   double mV[nTimeSteps];
   double mY[nTimeSteps];
   .....
   for (unsigned int t = 0; t < nTimeSteps; ++t){
     double currState = mY[t] ; // Backward dependency
     ....
     double logSpotPrice = func(currState, ...);
     mY[t+1] = logSpotPrice * A[t];
     mV[t+1] = logSpotPrice * B[t] + C[t] * mV[t];
     price[i][t] = logSpotPrice*D[t] +E[t] * mV[t];
   }
}
```

Same code runs on Intel Xeon and Intel Xeon Phi

The Fork-Join & SIMD layers

Fork-Join , Composable with Flow Graph

SIMD Layer

```
tbb::parallel_for(blocked_range<int>(0, nPaths, 256),
   [&](const blocked_range<int>& r) {
      const int block_size = r.size();
      double mV[nTimeSteps][block_size];
      double mY[nTimeSteps][block_size];
      ...
        for (unsigned int t = 0; t < nTimeSteps; ++t){

          for (unsigned p = 0; i < block_size; ++p)
          {
            double currState = mY[t][p] ;
            ....
            double logSpotPrice = func(currState, ...);
            mY[t+1][p] = logSpotPrice * A[t];
            mV[t+1][p] = logSpotPrice * B[t] + C[t] * mV[t][p];
            price[t][r.begin()+p] = logSpotPrice*D[t] +E[t] * mV[t][p];
          }
        }

}
```

```
for (unsigned i = 0; i < nPaths; ++i)
{
   double mV[nTimeSteps];
   double mY[nTimeSteps];
   .....
   for (unsigned int t = 0; t < nTimeSteps; ++t){
     double currState = mY[t] ; // Backward dependency
     ....
     double logSpotPrice = func(currState, ...);
     mY[t+1] = logSpotPrice * A[t];
     mV[t+1] = logSpotPrice * B[t] + C[t] * mV[t];
     price[i][t] = logSpotPrice*D[t] +E[t] * mV[t];
   }
}
```

Same code runs on Intel Xeon and Intel Xeon Phi

The Fork-Join & SIMD layers

Fork-Join , Composable with Flow Graph

SIMD Layer

```
tbb::parallel_for(blocked_range<int>(0, nPaths, 256),
   [&](const blocked_range<int>& r) {
      const int block_size = r.size();
      double mV[nTimeSteps][block_size];
      double mY[nTimeSteps][block_size];
      …
        for (unsigned int t = 0; t < nTimeSteps; ++t){
         #pragma omp simd
         for (unsigned p = 0; i < block_size; ++p)
         {
            double currState = mY[t][p] ;
            ….
            double logSpotPrice = func(currState, …);
            mY[t+1][p] = logSpotPrice * A[t];
            mV[t+1][p] = logSpotPrice * B[t] + C[t] * mV[t][p];
            price[t][r.begin()+p] = logSpotPrice*D[t] +E[t] * mV[t][p];
         }
        }

}
```

```
for (unsigned i = 0; i < nPaths; ++i)
{
   double mV[nTimeSteps];
   double mY[nTimeSteps];
   …..
   for (unsigned int t = 0; t < nTimeSteps; ++t){
     double currState = mY[t] ; // Backward dependency
     ….
     double logSpotPrice = func(currState, …);
     mY[t+1] = logSpotPrice * A[t];
     mV[t+1] = logSpotPrice * B[t] + C[t] * mV[t];
     price[i][t] = logSpotPrice*D[t] +E[t] * mV[t];
   }
}
```

Same code runs on Intel Xeon and Intel Xeon Phi

The Fork-Join & SIMD layers

Fork-Join , Composable with Flow Graph

SIMD Layer

```
#pragma offload_attribute(push, target(mic))
tbb::parallel_for(blocked_range<int>(0, nPaths, 256),
   [&](const blocked_range<int>& r) {
      const int block_size = r.size();
      double mV[nTimeSteps][block_size];
      double mY[nTimeSteps][block_size];
      ...
       for (unsigned int t = 0; t < nTimeSteps; ++t){
        #pragma omp simd
        for (unsigned p = 0; i < block_size; ++p)
        {
          double currState = mY[t][p] ;
          ....
          double logSpotPrice = func(currState, ...);
          mY[t+1][p] = logSpotPrice * A[t];
          mV[t+1][p] = logSpotPrice * B[t] + C[t] * mV[t][p];
          price[t][r.begin()+p] = logSpotPrice*D[t] +E[t] * mV[t][p];
        }
       }
}
#pragma offload_attribute(pop)
```

```
for (unsigned i = 0; i < nPaths; ++i)
{
   double mV[nTimeSteps];
   double mY[nTimeSteps];
   .....
   for (unsigned int t = 0; t < nTimeSteps; ++t){
     double currState = mY[t] ; // Backward dependency
     ....
     double logSpotPrice = func(currState, ...);
     mY[t+1] = logSpotPrice * A[t];
     mV[t+1] = logSpotPrice * B[t] + C[t] * mV[t];
     price[i][t] = logSpotPrice*D[t] +E[t] * mV[t];
   }
}
```

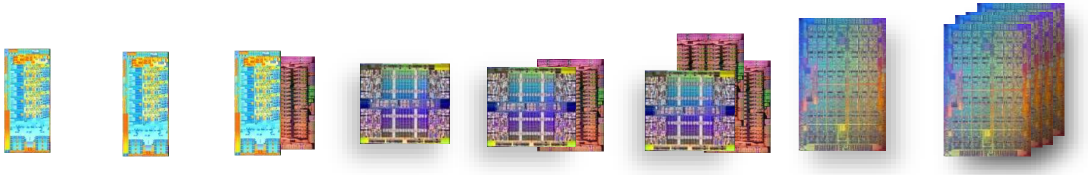
Same code runs on Intel Xeon and Intel Xeon Phi

Heterogeneous code sample from STAC-A2

```
#pragma offload_attribute(push, target(mic))
typedef execution_node  < tbb::flow::tuple<std::shared_ptr<GreekResults>, device_token_t >, double>
execution_node_theta_t;
…
void CreateGraph(…) {
…
theta_node = std::make_shared<execution_node_theta_t>(_g,
[arena, pWS, randoms](const std::shared_ptr<GreekResults>&, const device_token_t& t) -> double {
        double pv = 0.;
        std::shared_ptr<ArrayContainer<double>> unCorrRandomNumbers;
        randoms->try_get(unCorrRandomNumbers);
        const double deltaT = 1.0 / 100.0;
        pv = f_scenario_adj<false>(pWS->r, …, pWS->A, unCorrRandomNumbers);
        return pv;
    }
, true));
…
}
#pragma offload_attribute(pop)
```

Same code executed on Xeon and Xeon Phi, Enabled by Intel® Compiler

STAC A2:
Increments in HW architecture and programmability



	Intel Xeon processor E5 2697-V2	Intel Xeon processor E5 2697-V2	Intel Xeon E5 2697-V2 + Xeon Phi	Intel Xeon E5 2697-V3	Intel Xeon E5 2697-V3+ Xeon Phi	Intel Xeon E5 2697-V3+ 2*Xeon Phi	Intel Xeon Phi 7220	Intel Xeon Phi 7220 Cluster
	2013	2014	2014	2014	2014	2015	2016	2016
cores	24	24	24+61	36	36+61	36+122	68	68 x ?
Threads	48	48	48+244	72	72+244	72+488	272	488 x ?
vectors	256	256	256+512	256	256+512	256+2*512	512	2*512
Parallelization	OpenMP	→ TBB	TBB	TBB	TBB	TBB	TBB	TBB
Vectorization	#SIMD	OpenMP	OpenMP	OpenMP	OpenMP	OpenMP	OpenMP	OpenMP
Heterogeneity	N/A	N/A	→ OpenMP	N/A	OpenMP	→ TBB	N/A	TBB
Greek time	4.8	→ 1.0	→ 0.63	0.81	0.53	→ 0.216	0.22	???

1st Heterogeneous Implementation

Dynamic Load Balancing between 3 devices

Same user developed code

Summary

Developing applications in an environment with distributed/heterogeneous hardware and fragmented software ecosystem is challenging

- 3 levels of parallelism – task , fork-join & SIMD

- Intel TBB flow graph coordination layer allows task distribution & dynamic load balancing. Same user code base:
  - flexibility in mix of Xeon and Xeon Phi, just change tokens
  - TBB for fork-join is portable across Xeon and Xeon Phi
  - OpenMP 4.0 vectorization is portable across Xeon and Xeon Phi

Next Steps

Call For Action

TBB distributed flow graph is still evolving

We are inviting collaborators for: applications & communication layers

evgeny.fiksman@intel.com

sergey.vinogradov@intel.com

michaelj.voss@intel.com

THANK YOU FOR YOUR TIME

Michael Voss

michaelj.voss@intel.com

www.intel.com/hpcdevcon

Special Intel TBB 10th Anniversary issue of Intel’s The Parallel Universe Magazine

https://software.intel.com/en-us/intel-parallel-universe-magazine

CONTENTS

For more complete information about compiler optimizations, see our Optimization Notice.

Legal Disclaimer & Optimization Notice

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.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2016, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

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

intel®

intel®

experience
what's inside™