CnC Research Efforts

Zoran Budimlić

Rice University
Acknowledgments

Rice
  Michael Burke
  Vincent Cavé
  Philippe Charles
  Shams Imam
  Kath Knobe
  Vivek Sarkar
  Alina Sbîrllea
  Dragoș Sbîrllea
  Jianing Shi
  Sağnak Taşırlar
  Nick Vrvilo

Reservoir Labs
  Benoit Meister

Facebook
  Nicolas Vasilache

Micron
  Kyle Wheeler

UCLA
  Sara Achour
  Jason Cong
  Jens Palsberg
  Louis-Noel Pouchet
  Di Wu
  Peng Zhang
  Yi Zou

Intel
  James Brodman
  Sanjay Chatterjee
  Frank Schlimbach
  Kamal Sharma

Indiana
  Aaron Hsu
  Ryan Newton

UCSD
  Laura Carrington
  Pietro Cicotti

PNNL
  John Feo
  Ellen Porter
Topics

• CnC Flavors
• CnC Memory Management
• CnC Tuning
• Not covered in this talk:
  – CnC in classroom
  – Checkpoint/restart
  – Flexible preconditions
  – Hierarchy and reuse
  – Memory reuse
  – CnC for heterogeneous platforms
  – Applications
CnC Flavors
Two-level programming model

CnC, as a coordination language, is paired with a computation language
- Existing: C, C++, Haskell, Java, Scala, Python, Matlab
- In the works: Chapel, Fortran, JavaScript...

How is the CnC graph specified?
- Separate language, needs a translator to interface with computation language
  - Rice, UCLA, Indiana
- As computation language language API
  - Intel, Sandia
CnC-HJ

• Uses Habanero-Java runtime for step execution
• Steps are written in Java or Habanero-Java
  – Use of HJ allows for parallelism in step code
• Mature and tested, well suited to evaluating new CnC research ideas
• Work-sharing, work-stealing, data-driven runtimes
• Separate graph spec and translator

https://wiki.rice.edu/confluence/display/HABANERO/CNC-Download
CnC-HC

• Uses Habanero-C runtime for step execution
• Steps are written in C or HC
  – C wrappers can be used for CUDA, OpenCL, FPGA libraries
• Strict preconditions
• Work-stealing runtime
  – Heterogeneous execution, steps can be executed on CPU, GPU, FPGA
• Separate graph spec and translator with support for tag functions
Babel – language interoperability tool

- Language interoperability toolkit for high-performance computing
- Designed for fast, in-process communication
- Handles generation of all glue-code
- Mature project from LLNL

https://computation.llnl.gov/casc/components
CnC-Babel

- user writes textual description of CnC graph
- user annotates graph with extra information to be used by Babel
  - impl. language for Steps and Item Collections
  - types of items stored in the Item Collections
- user runs translator to generate code
  - Babel generates Step template
- user writes Step code and initialization logic
- user runs code using HJ
CnC-Matlab

• User writes CnC graph specification labeling steps which will be implemented in Matlab
• User writes Matlab code for steps
• User uses Mathworks’ Matlab Coder product to generate C code
• User writes glue code in C that manually
  • get/put items from/to collections
  • Convert SIDL arrays to Matlab arrays using provided helper functions
• User runs CnC translator to generate other glue code
• User compiles and runs application
CnC-Scala

- CnC-Scala has two modes for managing input dependences:
  1. Explicit “awaits” list for inputs
  2. Creation of continuations on get operations
- Implementation
  - Based on Habanero Scala (HS), implemented as a Scala library
    - uses compiler plugin for continuations support
    - details on HS can be found in OOPSLA 2012 paper
  - Only dependency is jsr-166y.jar (fork-join framework, already bundled with Java 7)

http://cnc-scala.rice.edu
Qthreads CnC

• C++ based CnC implementation developed at Sandia by Kyle Wheeler, Alina Sbirlea, and Dragos Sbirlea
  – Includes get-counts support
  – Three runtimes:
    • eager execution
    • strict preconditions,
    • flexible preconditions

• External API similar to Intel CnC
  – Support for Intel CnC benchmarks

• Based on Sandia Qthreads library
OCR (Open Community Runtime)
CnC-OCR

• CnC used as a high-level programming model
• Rely on a generic CnC runtime with explicit events to get natural mappings to:
  – Exascale architecture (simulators and real hardware)
  – Distributed CnC
  – Tuning through observation and adaptation
• Portability:
  – Programmer transition from CnC-HC to CnC-OCR is straight-forward
  – User is oblivious of the underlying runtime
• Potential for extensions:
  – Create extensions for monitoring applications
  – Attach debugging tools triggered by events
  – Enable speculative execution of steps
• Challenge:
  – Implement non-event-driven aspects of CnC runtime internals using a pure event driven approach

Thursday, Session 2, Talk 2: “CnC in an Event-Driven Programming Model”
CnC Memory Management
Memory management and CnC

A. Get-counts (Intel CnC 0.3)
B. Declarative collections (DAMP 2009)
C. Streaming CnC (DFM 2011)
D. Folding (Euro-Par 2012)
E. Bounded memory scheduling (PACT 2014)

Bounded memory scheduling:
A scheduling approach that ensures the parallel application does not use more memory than available.
Gives automatic memory collection too!
“The amount of memory required by a parallel program may be spectacularly larger than ... an equivalent sequential program.”

“Parallel memory requirements may be both large (relative to memory requirements of an equivalent sequential program) and unpredictable.”

Advantages of memory-aware scheduling

• State of the art schedulers do not adapt to available memory limitations.

Benefits of memory-aware scheduling:

- Avoid out-of-memory errors.
- Reduce disk swapping.
- Run on larger input sizes.
Bounded memory scheduling (BMS) problem

Given a program $P$ with input $I$ and a memory bound $M$, find a set of task ordering relations $TO$, such that every schedule that is legal for $P$ and also respects $TO$ has to fit in $M$.

Our solution:
The use of inspector/executor

• The solution is executing programs in two stages:

Inspector:
Query the computation for its structure.

Executor:
Based on the inspector, execute the program.
Dynamic Computation Graph

- Our programming model creates the dynamic computation graph without running the computation itself.

- We add functions mapping tags to:
  - Input and output data items
  - Prescribed tasks

- Limitation:
  - graph-level determinism.
  - only supports applications without data-dependent gets or puts.
BMS algorithm: Task ordering edges

- Task ordering edges ensure only the schedules that fit the memory bound are valid.

- Iff. A and B have the same color, we add ordering edges:
  - From the consumers of A
  - To the producer of B

These edges enforce non-overlapping lifetimes for same-color data.
BMS algorithm: Coloring heuristic

• Types of ordering edges:
  – transitive
  – serialization

• Ideal case: no serialization edges.
Results

Figure: Smith-Waterman results.

OpenMP uses pre-allocation, post-deallocation
- matches in performance,
- lags in memory

Figure: Cholesky Factorization results.

OpenMP exploits less parallelism
- worse performance
- lower memory footprint
Results: Using the sweet-spot

- Merge Sort: With 14% of parallel footprint, BMS offers 91% of the parallel speedup.

Figure: BMS-CnC executor time as a function of memory bound for Merge Sort.
CnC Tuning
CnC Tuning Overview

- Separation of Concerns

Application problem → Domain Program → Optimized mapping to platform

The domain expert:
- Finance
- Gaming
- Chemistry

The tuning expert:
- Parallelism
- Locality
- Load balancing

Domain expert doesn’t need to know a lot about parallelism

Tuning expert doesn’t need to know a lot about the app

- Domain expert, but later
- Different person, different skills
- Automated: static analysis / autotuning

CnC Tuning Model

• Two-level affinity model
  – Hierarchical affinity groups
    • compute/data affinity model (high-level)
  – Hierarchical Place Trees
    • machine affinity model (low-level)

• Two-level scheduling model
  – Tuning scheduler (staging area)
    • “tuning action” execution (tuning tree)
    • tuning actions use “tuning APIs”
  – Domain scheduler
    • task execution (hierarchical place tree)

• Goal: Achieve spatial and temporal locality
Domain / Tuning

Domain spec

• Deterministic (Easy to debug)
• Unified programming applicable for shared and distributed memory
• Efficient and scalable

Tuning spec

• Motivation
  – Semantics => partial ordering
  – some orderings more efficient than others
  – Distributed memory tuning is limited

• Semantic constraints from the domain spec are still requirements

• Tuning spec is a refinement of the domain spec
• The domain spec allows for many potential legal executions. Tuning avoids the bad ones.

One domain spec can have multiple tuning specs
Tuning Rician Denoising

Rician Denoising matrix size (7680x7680) and 30 iterations

- Up to 3.5x improvement on specific tile size
- Overall improvement of 19% over best optimal tiled version
Tuning for Distributed CnC

- Our goal is to achieve greater performance by optimal balance between resource utilization, load balance and communication
- CnC Model already has data and control methodology
- distCnC Model:
  - Provides control of work distribution
  - Provides controls of data distribution

CnC provides a unified programming model for shared and distributed memory
Distribution Function Generation

Given an application, generate a distribution function which minimizes execution time across a set of cluster nodes

• Distribution function value depends on steps instances (tasks) in an application and cluster node configuration
Approaches to Select Distribution Functions

• **Empirical Search (Dynamic)**
  - Exhaustively search for optimal distribution function
  - Pro: Takes all the environmental characteristics into account
  - Con: Time consuming
  - Sampling search space may help but still requires a significant amount of time

• **Analytical Model (Static)**
  - Creates an abstract model of the application and provides parameter output
  - Pro: Significantly less overhead
  - Con: approximation may produce performance inefficient results
  - Another approach is a general static analysis that works for all the programs

• **Expert Programmer**
  - Spends significant time to tune application
  - Relies on experience to tune it
  - Pro: May provide good solution based on experience with application
  - Con: Manual effort

• **Our Approach** combines analytical model with automated learning model
Our Approach

Step 1: Sample Data Points as training data across different distributions

Step 2: Generate the different parameters

Step 3: Learn the linear regression model variables

Step 4: Apply linear regression model across all points

Step 5: Choose Best Point with Minimum Execution Time Prediction
Dynamic Graph + Analytical Model Approach

CnC Program → CnC Trace File → Boost Spirit Parser → Dynamic Graph

Cluster Node Assignment (Distribution Function)

Total Work → Total Communication

Critical Path Algo (Topological Sort, EST, LFT)
Performance Distribution

Performance Distribution of Data Points for Cholesky Benchmark

% increase relative to best point vs Sorted Data Points
## Experimental Results

<table>
<thead>
<tr>
<th>Model</th>
<th>Predicted Actual Execution Time (secs)</th>
<th>Overall Best Execution Time (secs)</th>
<th>% Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Regression across all search points* (1536 points)</td>
<td>0.8428</td>
<td>0.7222</td>
<td>16.69% §</td>
</tr>
</tbody>
</table>

§ Range of % Difference is 0%-418%

* Trained across 150 Data points
Tuning for CnC-OCR

• Challenges:
  – OCR does not expose mechanisms to control location or task ordering in the way MPI or HC do
  – OCR has “hints” it can attempt to respect, or completely ignore
  – OCR runtime has its own ideas on where/when to put tasks and data, influenced by many factors (performance, power, energy, resilience)

• How do we translate CnC high-level tuning spec into low-level OCR hints with some reasonable expectations of performance?
Data Flow Graph Representation (DFGR)

• What it is:
  – Intermediate graph representation for macro-dataflow programs
  – Front-end for a compiler that targets heterogeneous architectures
  – Based on CnC

• What it offers: performance through analyzability
  – Enables the use of transformations on the application graph
  – Enables increased use of heterogeneity through static & dynamic scheduling
  – High-level view of applications
  – Easy programmability with high expressiveness using data-flow principles
  – Offer a framework for translating to parallel native code for various architectures
Final Thoughts

- Alive and kicking!
- Community effort
- Influence on other programming and execution models
- OpenCnC
- Exascale
- Many interesting research directions in languages, compilers, runtimes, tuning
Backup Slides
CnC Babel Runtime

- Step Instances (server)
  - Item Collection (Data)
- Step Instances (client)
  - Control Collections (Scheduling)
  - Item Collection (Scheduling)
- HJ-CNC Runtime

Languages supported: Java/Python/C/C++/Fortran
Checkpoint/Restart for CnC

“what’s good for simplifying parallelism is good for simplifying resilience”*

• Pushing CnC for extreme-scale programming

• CnC has many interesting properties:
  – Execution graphs
  – Single-assignment data
  – Referentially transparent computation steps
  – Execution frontiers
  – Hierarchical graphs

• How to leverage for CnC checkpoint/restart?

*Inter-Agency Workshop on HPC Resilience at Extreme Scale. 2012.
Checkpoint/Restart in CnC

• Execution Frontier (XF)
  – Currently-active subset of CnC graph state
  – Works as input/output of a graph

• CnC Checkpoint
  – Checkpoint is a snapshot of the XF
    • Might be out of sync
    • Could be in different format (e.g. compressed)
  – Derived attributes handled separately in the checkpoint
    • Data might arrive out of order
    • Don’t want to mark a step as data ready if not all the data has shown up in the checkpoint!
CnC in Classroom
CnC in Classroom

• UCLA CS133 – Parallel Programming (Spring 2013)
  – Upper-division undergraduate class
  – 75 students (junior and senior)
  – Lecturer: Jason Cong, TA: Di Wu
    • Vivek Sarkar gave two guest lectures on CnC and shared Rice CnC C++ implementation
    • Alina (Rice) gave a guest lecture the discussion session about program details

• Class project
  – MI pipeline acceleration
  – Three applications: MRI reconstruction, denoise, segmentation
  – Four parallel languages: OpenMP, MPI, OpenCL, CnC
  – Each student needs to choose application and implement in all four languages
  – CnC eventually made as an optional bonus, given difficulty faced by the students
    – 11 students finished (14.7%)

• Choice of project is not optimal
  – Each application is highly iterative
• Students who completed the CnC project enjoyed it—quote from one student
  – ...I thought the CNC implementation was definitely the most interesting part of the class. I think the students who skipped it definitely missed out.
  – Working out all the problems that arose (at least for me) from both not fully understanding exactly how the graphical nature of CNC was put together, and just trying to figure out what each different part of code was actually doing, took me a really long time. I think it took me upwards of 6 or 8 hours to get a super simple 2D array addition operation working.
  – But that was definitely a helpful experience, and while I totally understand that there's no way many of the students could devote that much time in a single afternoon just to get a very basic sample up and running, that's the aspect of this class that I think was more useful than many of the other ones I've taken. I can't help but think if everyone else had to go through that, instead of it being optional, would they have gotten as much out of it as I did?
  – I would like to mention here that Di Wu did an amazing job at getting a grasp on CNC very quickly so he could help the rest of us with our stuff. He also mentioned spending several hours getting simple samples up and running for the rest of us, which was absolutely great. He was also great getting our question forwarded onto Alina and helping us out with getting around some of CNCs restrictions to better use it with the project itself.
Students Feedback (Cont’d)

• Non-intuitive programming model
  – Not as easy as OpenMP
  – Requires detail knowledge of the application

• Not able to handle in-place operations
  – Data collection not freed in graph
    • Need to maintain ‘MaxIter’ images in memory
    • Memory overflow
  – Destroy graph each iteration
    • Too expensive

– Solution
  • ‘hacking’: use CnC to create steps, and use ‘finish’ to enforce explicit synchronization

```c
for (iter=0; iter<MaxIter; iter++)
{
    // processing 2d image
}
```
Applications
Applications

• Dense linear algebra: Cholesky (everyone 🙏), Eigensolver (GA Tech)
  – Best Paper award, IPDPS 2010
• Medical imaging applications (UCLA, Rice)
  – Combining CPU, GPU and FPGA steps for performance/energy
• Smith-Waterman (Rice)
• Unbalanced Tree Search (Rice)
• Graph 500 (Rice)
• Floyd-Warshall (Intel)
• LULESH (LLNL, Intel)
• CoMD (LANL)
• RTM 3DFD Stencil (Intel)
• Animation (Dreamworks)
LU Decomposition in CnC-Scala

- 3 Item Collections, 5 Tag Collections, and 8 Steps.
Python

• Multithreading is not an option
  • The Global Interpreter Lock serializes computations
  • Not all extension modules support multithreading
  • Jython and IronPython do not support all extension modules

• Multiprocessing module
  • Requires explicit launching of processes/jobs
  • Blocking synchronization on job completion for data
CnC-Python Build Model

CnC Graph Spec (text)

Generated Stub code (Habanero-Java and Babel)

- Step code (in Python)
- main() (in Python)

make build (from generated Makefile)

Compiled code

env CNC_NUM_WORKERS=W make run
(from generated Makefile)

Output
CnC-Python Implementation

• Targeted to shared-memory multiprocessors

Java-based runtime task scheduler (in Habanero-Java)

Used for Java-Python interoperability
Comparison with Parallel Python

SumPrimes Benchmark from Parallel-Python

<table>
<thead>
<tr>
<th># of Workers</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Python</td>
<td>125.63</td>
<td>63.61</td>
<td>42.24</td>
<td>32.19</td>
</tr>
<tr>
<td>CnC-Python</td>
<td>33.86</td>
<td>11.54</td>
<td>6.88</td>
<td>5.41</td>
</tr>
<tr>
<td>Speedup Factor</td>
<td>3.71</td>
<td>5.51</td>
<td>6.14</td>
<td>5.95</td>
</tr>
</tbody>
</table>

Execution Time (in secs)

Number of Workers

- CnC-Python
- Parallel-Python
Matlab

• Widely used by domain experts
  • For numerical/scientific computing
  • Offers concise simple syntax
  • Excellent documentation
• Helps in fast prototyping
  • but sometimes executes slower
CnC-Matlab Build Model

CnC Graph Spec (text)

Generated Stub code (Habanero-Java and Babel)

cnc_translate

- Step code (in Matlab)
- main() (in Matlab)

Matlab Coder

- Generated code (in Matlab Coder)
- Step wrapper code (in C)
- main() wrapper code (in C)

make build (from generated Makefile)

Compiled code

env CNC_NUM_WORKERS=W make run (from generated Makefile)

Output
CnC-Scala Build Model

CnC Graph Spec

cnc Scala translate

Generated Stub code

cnc Scala compile

Compiled bytecode

cnc Scala run

Output

- Step code
- main() method
Continuations

- Represents rest of the computation from a given point in the program
- Allows
  - suspending current execution state
  - resuming from that point later
- We need only delimited one-shot continuations
  - Scala has shift-reset!
- Benefits of continuations
  - No re-execution of code
  - Allow arbitrary (data-dependent) gets
  - Threads never block
    - No extra threads created
CnC-Scala Runtime - get

- If item is available return it
- Else store continuation

```scala
get(tag: TagType): ItemType = {
  if (itemAvailable)
    return item
  else
    shift { continuation =>
      // store continuation
      // when cont resumes, item is available
      return item
    }
}
```
CnC-Scala Runtime - put

- Store item
- Resume waiting continuations

```scala
put(tag: TagType, item: ItemType) {
  // store item into DDF
  // resume **ALL** continuations waiting on item
}
```
Runtime Policies

• **Blocking Gets**
  – Step starts to execute when prescribed
  – If a Get fails, the thread running the step is suspended
  – Can only be done in a work-sharing runtime
  – Very poor scaling

• **Rollback and Replay (RR)**
  – Steps start to execute when they are prescribed
  – If a Get fails, step is killed
  – Step is restarted by the step doing a Put on the data that Get failed on.

• **Data Driven (DD)**
  – Steps do not start to execute until all data is available
  – Dependencies are “filled in” when step is prescribed
  – Once all dependencies are satisfied, step executes => Gets are ensured to succeed.

• **Work-first**: The worker thread executes the prescribed step immediately, allowing the execution of the currently running step to be stolen by other worker threads

• **Help-first**: The runtime puts the prescribed step on a queue to be stolen by other worker threads
Comparing Runtime Policies

- ForkJoin benchmark v1, “no-work”, 100k-1M tasks
Comparing Runtime Policies

- ForkJoin benchmark v2, 100 Gets, 100k-1M tasks
Comparison of Runtime Policies

• The DD Runtime performs better under heavy load of small parallel tasks
  – Avoids the task-creating overhead
• The RR Runtime performs better for a heavy number of successful Get calls
  – Eagerly executes tasks, thus obtaining more parallelism than the “safe” approach
• Work-first policy is more efficient than help-first for nested fork-join programs (e.g., Cilk, Habanero-C)
• Help-first policy is more efficient than work-first for parallelism expressed with a parallel for loop (detailed study: [1])
• For “normal” sized tasks the two runtimes
  – Scale well
  – See little differences between each other

Dynamic Single Assignment (DSA) Folding

- Problem: it is difficult to automatically identify when CnC items are dead
  - Cause: item keys can be recomputed, unlike general references
  - User intervention may simplify the task and offer the opportunity for further optimization

• Transform the item keys with a *folding function*, which maps logical (dynamic single assignment) keys to fewer folded (multiple assignment) keys.

\[
\text{Fold(DSA key)} = \text{folded key}
\]

\[
|\text{DSA Keys}| > |\text{folded Keys}|
\]
Folding Function for Routing Benchmark

• The folding function implements a generalized version of the two buffer approach

```c
point fold(point key) {
    // point = [repetition, iteration, nodeId]
    int i = p.get(0);
    int j = p.get(1);
    int k = p.get(2);
    point fKey = point(i%2, j%2, k);
    return fKey;
}
```
Results for Routing Benchmark

Routing with unreliable links

[Graph showing results for different benchmarks with lines for different conditions]
CnC Program State

Execution Graph

- Control Tag
  - Avail

- Data Item
  - Avail
  - Data

- Step
  - Control Ready
  - Data Ready
  - Executed

Diagram by Kath
Runtime Research

- Runtime policies
- DSA Folding
- Checkpointing/restart
- Flexible Preconditions
- Distribution function generation
- CnC Tuning
Flexible Preconditions

• Eager task creation
  – Tasks are spawned as soon as they are prescribed
  – Once they start to execute, they may encounter an unavailable dataflow dependence

• Strict preconditions
  – Dataflow dependences must be declared before tasks start running
  – Limiting expressiveness

• Flexible preconditions
  – allow partial specification of the preconditions of tasks
  – performance and memory behavior of strict preconditions
  – programmability of eager spawning
Influence on other languages/efforts

- CDSC-GL
- Habanero-C
- OCR
Data-driven Futures and Data-driven Tasks in Habanero-Java and Habanero-C

Habanero-C syntax:

\[ \text{DDF}_t^* \text{ ddfA} = \text{DDF}_\text{CREATE}(); \]
Allocate an instance of a data-driven-future object (container)

\[ \text{async IN (ddfA, ddfB, ...)} \text{ AWAIT(ddfA, ddfB, ...)} <\text{Stmt}> \]

– Create a new data-driven-task to start executing Stmt after all of ddfA, ddfB, ...
  become available

\[ \text{DDF.PUT(ddfA, V)}; \]

- Store object V in ddfA, thereby making ddfA available

- Single-assignment rule: at most one put is permitted on a given DDF

\[ \text{DDF.GET (ddfA)}; \]

- Return the value stored in ddfA (non-blocking)

- Safely done by asyncs that have ddfA in their await clause

- CnC item collections are just hashmaps of DDFs
- CnC steps are DDTs
- CnC-HJ uses DDFs and DDTs as its default runtime
Summary of OCR Open Source Project

• Hosted on 01.org
• Goals
  – Modularity
  – Stable APIs
  – Extreme flexibility in implementation
  – Transparency
• Development process
  – Continuous integration
  – Quarterly milestones
  – Mailing lists for technical discussions, build status, etc
• Organization
  – Steering Committee (SC) --- sets overall strategic directions and technical plans
  – Core Team (CT) --- executes technical plan and decides actions to take for source code contributions
  – Membership of SC and CT will turn over periodically based on level of participation

https://01.org/projects/open-community-runtime
Affinity and Cancelation (new in 0.9)

**Intel**

- **Affinity**
  - Like distribution (tuner::compute_on(tag)) tuner::affinity(tag) defines preferred thread for executing given step
  - Runtime option allows pinning threads to cores configurable stride for the mapping

- **Cancellation**
  - Second feature (after unsafe_get) which allows non-determinism
  - Tuner can cancel execution
    - Best effort policy, steps might still execute
    - In particular cancelation might come after execution
  - Customizable interface
    - CnC::cancel_tuner provides cancel(tag) and cancel(all)
    - Other strategies possible, like after system inspection or so
CnC on Intel® Xeon Phi™

**Intel**

- Preliminary experiments with 3dfd-TTI
  - 2 x 73-point stencil
  - Used in seismic exploration
- Strong NUMA effects on Xeon
  - “Auto-tunes” with 2 processes on a 2-socket box
- Pinning needed on Phi to achieve good performance
- Adding a Phi (same code) achieves 30% speedup
- Issues with clusters
  - Might be caused by MPI implementation
- Note: top-notch native (Fortran) Phi(-only) implementations can be almost as fast as this Xeon/MIC combo
  - With special configuration for a given data set
Scalability

**Intel**

- 3dfd-TTI (2 x 73-point stencil)
- Quasi-ideal weak scaling
- Tradeoff between tile-size and communication overhead
- Strong scaling
  - 47% efficiency despite 4x more tiles than cores
  - Asynchronous execution!

**RTM-3dfd parallel efficiency (1024^3)**

2 processes per node, tiles:128x64x256

- ISO/C++-weak
- TTI/C++-weak
- TTI/C++-strong

1 tile per core

1 tile per 4 cores
**Current work: Combining CnC and MPI**

*Intel*

- Exploring adding CnC to existing HPC/MPI codes
  - An evolutionary alternative to the current revolutionary approach
- Current idea: support embedding CnC blocks/phases into an MPI application
  - Allows for adding or converting pieces of a code, in contrast to an all-or-nothing scenario
  - Each CnC phase works on a subset of processes
    - Specified by an MPI-Communicator
    - Multiple concurrent but independent CnC phases are possible
  - Each CnC phase can have
    - Either a single environment (single-process view)
    - Or a shared environment (SPMD-style)
  - Initial prototype implemented
- Now looking for collaborators and applications
Current work: CnC.js

Intel

• Exploring CnC for JavaScript
  – JavaScript community searching for a solution to general parallelism
    • One JS requirement is race-freedom, which CnC guarantees by design
  – API on the JS level (e.g. as a language construct)
    • Enforces CnC graph (e.g. no write on consume-only edges, access only via declared relations)
  – Experimenting with a prototype in spidermonkey/firefox
    • Basic API works single-threaded (with the C++ runtime under the hood) e.g. in a face-detection app
    • Enforces DSA (very invasively: deep-freezes everything)
  – Next
    • Get parallelism
      – Not trivial: parallel execution simply not foreseen in the complex (and huge) runtime
      – Exploring acceptable limitations
    • Disallow side-effects (also more difficult than anticipated)
    • Tag functions
    • Discuss with stakeholders?
Open Community Runtime (OCR)

- A runtime framework that ...
  - is representative of execution models expected in future extreme scale systems
  - can be targeted by multiple high-level programming systems
  - can be effectively mapped on to multiple extreme scale platforms
  - can be extended and customized for specific programming and platform needs
  - can be used to obtain early results to validate new ideas
  - is available as an open-source testbed

- Strongly influenced by the CnC model
  - decomposition of algorithm into steps/items, tuning, event-driven execution
CDSC-GL

Rice, UCLA

- Context: NSF Expeditions Center for Domain-Specific Computing (CDSC) --- UCLA, Rice, OSU, UCSB
- CDSC-GL serves an intermediate graph language used in mapping
- CDSC-GL can be generated from higher-level programming systems, and can operate on both graph and non-graph data
- CDSC-GL can be mapped on to heterogeneous hardware (including FPGAs) with synchronized access to non-graph data
- CDSC-GL will need to be hierarchical to support mapping on to hierarchical machine models
  - Design of hierarchy in CnC is currently in progress; will influence design of hierarchy in CDSC-GL

Tuesday, Session 3, Talk 4: “CDSC-GL: A CnC-inspired Graph Language”
CnC 0.9 Release

Intel

- API implementing full CnC semantics
- Shared and distributed memory
- Separate interfaces for domain and tuning
- Affinity, Cancelation
- Support for Intel® Xeon Phi™
- Tuner for CnC::parallel_for
  - switch on/off checking dependencies, priority, affinity, depends and preschedule
- New examples (floyd-warshall, nqueens, dedup, UTS)