Designing HPC Software for an Uncertain World of Hardware

Wael R. Elwasif & David E. Bernholdt

Oak Ridge National Laboratory
”Predictions are hard, especially about the future”
Yogi Berra (Maybe) :-)

"Reports that say that something hasn't happened are always interesting to me, because as we know, there are known knowns; these are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns -- the ones we don't know we don't know." Donald Rumsfeld

“It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in 5 years.” (John Von Neumann, circa 1949)
Caveats

- There is little or no ESM content in this talk.
- This is not a talk aimed at selling you one or more tools for doing your work.
- This talk IS about *lessons learned*, and the *abstractions*, *approaches*, and *techniques* that may be brought to bear on the coupling problem.
- You probably know this stuff already.
The Changing Hardware Landscape

- Petascale Now
  - Exascale in 10 years
- Do we care? **YES**
- Trickle-down computing
  - 2010’s national supercomputer is
  - 2013’s campus resource, is
  - 2016’s group cluster, is
  - 2019’s personal server

*(Robert Harrison)*

*Maybe THIS transition will be similar to the ones we had before – maybe not!!!*
What is different this time?

- Massive scale
  - (current) MPI probably not enough.
  - Shrinking memory/core.
  - Shrinking MTBF.
  - Unforeseen failure modes.
- Heterogeneity
  - Performance & Power
  - Unification down the road??
- Multilevel parallelism
  - More than just MPI/OpenMP
  - Need new programming models? Languages?

*Rewriting code beyond simple porting??*
Future-Proof Code ??

- Three intertwined code pillars
  - Architecture
    - What parts make my code?
    - How do these parts relate to one another?
  - Composition
    - How do I put these parts together?
    - How to overcome points of impedance mismatch.
  - Generation
    - What to write myself, and what to use a tool/DSL for?
    - How to integrate generated code with hand-written parts?
- Nothing novel, but we will need
  - User accessible tools
  - Target new bottlenecks
  - Accept some performance penalty
Design For Change

- Minimize disruptions when a sub-part evolves/changes.
  - Rethink the notion of "Globals".
- Algorithms, data structures will probably change
  - Maybe multiple times as the hardware evolves.
- Manage inevitable increase in code complexity
  - Higher fidelity simulations, novel scalable algorithms, multilevel parallelism, ... etc.
- Facilitate contributions and community participation
  - Testing, Validation, Standard Compliance, ..etc
- Long code lifetime
  - Outlast architects, contributors, and platforms
- Increased decentralized development
  - Enable effective distributed collaboration and contributions
Components: Building Blocks for Robust Large Scale HPC Code

• Basic Idea:
  • Extend object-oriented ideas (abstraction, encapsulation, separation of concerns, modularity), with **composition**

• Opportunities
  • Effective functional partitioning of large scale applications.
  • Compartmentalize change as the hardware evolves.
  • Provide a handle on code complexity

• Challenges
  • **Hostile** HPC environment (parallelism, performance, languages)
  • Decentralized decision making (mostly)
10 Years of The Common Component Architecture (CCA)

- Component architecture for the HPC world
  - Address issues with commodity component environments
- HPC relevancy
  - Legacy software support
  - Parallel and distributed computing
  - Languages C, C++, Fortran, Java, Python
  - Data structures.
- One small step towards better HPC codes
  - More work yet to be done
Basic CCA Concepts

• **Components**
  – Are units of software development/functionality
  – Interact only through well-defined interfaces
  – Can be composed into applications based on their interfaces

• **Ports**
  – Are the interfaces through which components interact
  – Follow a provides/uses pattern
  - *Provided* ports are implemented by a component
  - *Used* ports are functionality a component needs to call

• **Frameworks**
  – Hold components while applications are assembled and executed
  – Control the connections of ports
  – Provide standard services to components

*Screenshot of application in the Ccaffeine framework’s GUI*
Food For Thought

• Can we all agree on what a *component* is?
• Even if we do not get seamless re-use and “plug and play”, component remain useful for effective management of increasing code complexity.
• How do we go about fixing ports and interfaces
  • Community consensus?.
  • Adopt the ones that are “more complete”, “more flexible”, “better documented”, “better advertised”, ...
• How much investment are willing to make into “componentization” of the codes?
  • While not immediately producing new science.
**Babel** is a Foundational Part of the CCA

- Babel provides
  - Fast inter-language communication for C, C++, Fortran, Python, and Java
  - Consistent OO type system
  - Encapsulation
  - Remote Method Invocation (RMI)

- Babel’s SIDL supports:
  - Language independent specifications
  - Interface contract specifications
Contracts: Runtime Checkable Annotations in SIDL Specifications

Ensure the following (with respect to the specification) at runtime:

- Components written correctly
- Programs use components correctly
- The proper component is used (at the right time)

```cpp
double dot (in array<double> u, in array<double> v, in double tol)
throws sidl.PreViolation, sidl.PostViolation;
require /* Preconditions */
  u_is_1d: (u != null) implies (dimen(u) == 1);
  v_is_1d: (v != null) implies (dimen(v) == 1);
  same_size: size(u) == size(v);
  non_negative_tolerance: tol >= 0.0;
ensure /* Postconditions */
  areEqual(u, v, tol) implies (result >= 0.0);
  (isZero(u, tol) and isZero(v, tol))
    implies nearEqual(result, 0.0, tol);
```

**Preconditions** are obligations on component callers

**Postconditions** are obligations on component implementations
Food For Thought (2)

• (Generalized) Language interoperability is not easy.
• Yet we are likely to need it for the foreseeable future.
  • Or can we assume that C(++) + Fortran 2XXX is all we will ever need ??
• Next challenge: *Programming Model Interoperability*
  • Beyond just MPI + OpenMP
  • Think Message Passing + Threads + PGAS + GPGPU + ....
• Contracts can make your code robust, but think about
  • Automated generation of associated code.
  • Performance overhead (sampling and selective enforcement can help)
The CCA and The Three Code Pillars

- Architecture
  - Methodology for partitioning HPC code
- Composition
  - Assembling components into applications
- Generation
  - *A LOT* of generated glue code (mainly language interoperability - Babel)

- BUT
  - All or nothing approach raises adoption cost and entry barrier.
  - Many concepts embedded in a single implementation.
  - Need something better (more on that later).
Code (and other artifacts) Generation

- Can no-longer leave all generation to compilers
  - Well, we still need them at the end of the tool chain.
  - Higher level (better ?) languages stuck in acceptance/deployment/ legacy limbo (Chapel, X10, Fortress)
  - Certain constructs lend themselves naturally to formal high level representation
- Domain Specific Languages (DSL)
  - Address semantic gap between high level thinking and low level common programming languages.
  - Can be simple or complicated (based on the domain).
  - May require significant CS investment, BUT
  - Pay off can be significant
Getting DSLs Right

- The Domain?
  - Too broad? Too narrow? Or Just right.
- Level of abstraction in the language?
  - Too high? Too low? Or just right.
- Language constructs?
  - Too simple? Too complicated? Or just right.

Words heard around here:

*Generation, Cupid, Coupler Generation, BFG, XML transformations, XSD, Code Generation, ...*

**DSLs done right can have A HUGE impact on the application**
The Tensor Contraction Engine: A tool for Quantum Chemistry

Oak Ridge National Laboratory
David E. Bernholdt, Venkatesh Choppella, Robert Harrison

Pacific Northwest National Laboratory
So Hirata

Louisiana State University
J Ramanujam

Ohio State University
Gerald Baumgartner, Alina Bibireata, Daniel Cociorva, Xiaoyang Gao, Sriram Krishnamoorthy, Sandhya Krishnan, Chi-Chung Lam, Quingda Lu, Russell M. Pitzer, P Sadayappan, Alexander Sibiryakov

University of Waterloo
Marcel Nooijen, Alexander Auer
High Accuracy Quantum Chemistry Methods

- Coupled Cluster Methods are widely used for very high quality electronic structure calculations
- Typical Laplace factorized CCSD(T) term:

\[
A3A = \frac{1}{2} \left( X_{ce,af} Y_{ae,cf} + X_{ce,af} Y_{ae,cf} - X_{ce,af} Y_{ae,cf} + X_{ce,af} Y_{ae,cf} \right)
\]

\[
X_{ce,af} = t_{ij}^{ce} t_{ij}^{af} \quad Y_{ae,cf} = \langle ab|ek\rangle\langle cb|fk\rangle
\]

- Indices: \( i, j, k \) \( O(O=100) \) values, \( a, b, c, e, f \) \( O(V=3000) \)
- Term costs \( O(OV^5) \approx 10^{19} \) FLOPs; Integrals \( \sim 1000 \) FLOPs each
- \( O(V^4) \) terms \( \sim 500 \) TB memory each

\[
\Phi_{GW} = \frac{1}{2} - \frac{1}{2} - \frac{1}{4} - \frac{1}{6} - \frac{1}{8} - \ldots
\]

Hartree  Fock  Infinite chain of dressed electron-hole bubbles
CCSD Doubles Equation

\[ h\bar{a}[b,c]^i \equiv \text{sum}[f[b,c]^j][a,b,c][i] - \text{sum}[f[k,c]^i][k,b]^j[a,b,c][i] + \text{sum}[f[a]^i][i,j][b,c][a,b][j] \]
A High Level Language For Tensor Contraction Expressions

range $V = 3000$;
range $O = 100$;

index $a,b,c,d,e,f : V$;
index $i,j,k : O$;

$\text{mlimit} = 10000000000000$;

function $F1(V,V,V,O)$;
function $F2(V,V,V,O)$;


begin

$X = \sum \sum [F1(a,b,f,k) * F2(c,e,b,k), \{b,k\}] * \sum [T1[i,j,a,e] * T2[i,j,c,f], \{i,j\}]$, \{a,e,c,f\};

end

$A_3A = \frac{1}{2} X_{ce,af} Y_{ae,cf} + X_{ce,af} Y_{ae,cf} + X_{ce,af} Y_{ae,cf} + X_{ce,af} Y_{ae,cf}$

$X_{ce,af} = t_{ij}^{ce} t_{ji}^{af}$

$Y_{ae,cf} = \langle ab|ek\rangle \langle cb|fk\rangle$
**TCE Elements**

- **Algebraic Transformations**
  - Minimize operation count
- **Memory Minimization**
  - Reduce intermediate storage
- **Space-Time Transformation**
  - Trade-offs between storage and recomputation
- **Storage Management and Data Locality Optimization**
  - Optimize use of storage hierarchy
- **Data Distribution and Partitioning**
  - Optimize parallel layout
TCE Impact on Productivity

- Drastic reduction in time to develop CCSD code

<table>
<thead>
<tr>
<th>Theory</th>
<th># Terms</th>
<th># F77Lines</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCD</td>
<td>11</td>
<td>3209</td>
<td>1978</td>
</tr>
<tr>
<td>CCSD</td>
<td>48</td>
<td>13213</td>
<td>1982</td>
</tr>
<tr>
<td>CCSDT</td>
<td>102</td>
<td>33932</td>
<td>1988</td>
</tr>
<tr>
<td>CCSDTQ</td>
<td>183</td>
<td>79901</td>
<td>1992</td>
</tr>
</tbody>
</table>

- More than 25 methods implemented.
- Incorporated into NWChem.
- O(1M) lines of generated code (and growing).
Bocca: Tool & Language for Rapid CCA Component Development

• Motivation
  • Steep learning curve for generating CCA components
  • Complex build system
    – Babel generates many files, different language.
    – Too many dependencies.
  • A lot of work before you get to insert your logic into a component.

• Approach
  • Enable rapid generation of complete CCA compliant artifacts
    – Components, Ports, Classes, Interfaces
  • Command line development environment for interaction with managed code
  • Auto-generated build system that manages the entire project.
Bocca In action

#!/bin/bash
bocca create project proj
cd proj
./configure --with-languages='f90 cxx'
bocca create port Solver
bocca create component Newton -l cxx 
   --provides Solver@SolverPort
bocca create component Driver -l f90 
   --uses Solver@SolverPort\n   --go Compute
make
make testgui

real  0m55.078s
user  0m34.784s
sys   0m8.514s

- 48 Directories
  - 17 BOCCA internal
- 134 Files
  - 2 User-level cxx files
  - 2 User-level F90 files
Food for thought (3)

- DSL works IFF co-developed by the domain experts AND programming language experts
  - Otherwise, we'll probably not get it “right”
- Expand the notion of Domain
  - ESM? Coupling? ... 
- Tools, DSL and Libraries, which is a better fit for the problem on hand.
Some Lessons Learned from 10 Years of the Common Component Architecture

- The CCA component model specification is well defined and stable.
- Associated CCA tools being used in several domains.

BUT

- Pervasive changes to existing codes are hard
  - On-Ramp CCA project to semi-automate this process
- Tight inter-dependence of CCA tools and features
  - All-or-nothing vs a-la-carte tools and techniques
- Component approach is useful, but sometimes not absolutely necessary for improved software composition
  - Need a “pay for what you use” approach.
Next Generation HPC Composition: COMPOSE-HPC

• On-ramp (semi-) automatic componentization prototype.
  • Using source-to-source (s2s) transformations and code annotation
• Why not generalize?
  • Annotations can guide transformations while retaining original expression and functionality of source code
  • S2S tools can effect many kinds of transformations
  • Some optimizations require more context than compilers have.
• Grand vision: HPC environment is changing so rapidly that we need customizable, language-aware tools to effectively manage change
  • Today's "language tools" are not accessible to most developers.
  • Move from ad hoc solutions to systematic reproducible solutions.
• Software Composition as the overarching theme.
Addressing Challenges to Software Composition (Broadly Interpreted)

- Distributed development teams
  - Geographical
  - Scientific discipline

- First-party and third-party modules

- Different degrees of verification and validation

- Different programming languages

- Different processors (heterogeneity, accelerators, etc.)

- Approaches to parallelism
  - Threading models
  - Multiple-program multiple data execution
On-Ramp: Transforming Annotated Source Into CCA Components

1. Developer annotates original code

OnRamp instructs Bocca to create empty components

2. Application broken into components automatically

3. OnRamp pastes application functionality and necessary glue code into the empty components

4. Connect components together to create an application-specific framework reproducing the original application functionality

```c
// File: main.c

// %CCA COMPONENT id=TestComp
// %CCA PORT id=TestPort

// %CCA BEGIN
float add2(float a, float b) {
    return a + b;
}
float add3(float a, float b, float c) {
    return add2(a, b) + c;
}
int main(int argc, char **argv) {
    float x = 1.0;
    float y = 2.0;
    float z = 3.0;
    float n = add3(x, y, z);
    return (int)n;
}
// %CCA END
```
Annotations to Guide Transformations

/* %CCA PORT id=Array1DPort
   BEGIN */

/* %CCA ARRAY elements=x length=len */

```c
int Sum(int len, int *x) {
    int i = 0, sum = 0;
    for (i=0;i<len;i++) sum += x[i];
    return sum;
}
```

User’s code is kept intact, but ready for automatic transformation for use in other environments. Long history in HPC; currently used in OpenMP, HMPP, etc.
The “Language Tools” Landscape

Today

- Single-purpose
  - OpenMP
  - HMPP
  - PGI Accelerator
  - ...

- Refactoring
  - Eclipse CDT (C, C++)
  - Photran (Fortran)
  - ...

- Fixed capabilities
  - (Application) developer-friendly

- Source-to-source
  - ROSE
  - ...

- Rewriting engines
  - Maude
  - Stratego
  - ...
  - ...

- Open-ended capabilities
  - Requires deeper computer science concepts
Refactoring

- Code refactoring is the process of changing a computer program's source code *without modifying its external functional behavior* in order to improve some of the nonfunctional attributes of the software
  - Improve readability, reduce complexity, restructure architecture or object model
- Refactoring tools widely used in industry
- In HPC refactoring is usually done manually or sed scripts
  - Intimidating, so rarely done
  - Error prone
- Photran refactoring for Fortran
  - Rename, Encapsulate variable, Interchange loops, Introduce Implicit None, Move Saved Variables to Common Block, Replace Obsolete Operators, Standardize Statements, Remove Unused Variables, Data to Parameter, Extract Procedure, Extract Local Variable, Canonicalize Keyword Capitalization, Make COMMON Variable Names Consistent, Add ONLY Clause to USE Statement, Minimize ONLY List, Make Private Entity Public

Source-to-Source Transformation

- Refactoring is a source-to-source (s2s) transformation of the source code
- Like a compiler, but…
  - Refactoring transformations rather than optimizations
  - Emits code in a programming language instead of object code
- ROSE is a compiler infrastructure designed for s2s
  - Led by Dan Quinlan (LLNL), DOE supported
  - Primary use is for performance optimizations
  - Also used for automatic differentiation
  - Writing transformations is hard

```c
int main() {
    Range I(1,98,1), J(1,98,1);
    doubleArray A(100,100);
    doubleArray B(100,100);
    A(I,J) = B(I,J)+B(I,J)+B(I,J)+B(I,J);
    return 0;
}
```
A Grand Vision for Software Evolution in a Rapidly Changing World

- Leverage language tools to automate transformations to facilitate software evolution
- Systematically transform and take new version forward

Or…

- Support multiple platforms and environments using annotated source code and reproducible build-time transformations
KNOT: Nimble Orchestration Toolkit (1/2)

- Building on various existing language tools, focusing on developer-friendliness, generality, and flexibility
- PAUL: annotation parsing facility
- ROTE: Retargettable Open Transformation Engine
KNOT: Nimble Orchestration Toolkit (2/2)

- BRAID: code generation and optimization
  - Language interoperability and similar problems
- ScidADL: architecture description language
KNOT Scenarios in development

- Automatic Performance Instrumentation
  - Exemplar of simple annotation and transformations
- Simplifying use of GPU-based accelerators
  - Auto-generate tedious and error-prone “plumbing” code.
  - e.g. data marshaling between host and device.
  - Exemplar of more complex transformation
- Interface contracts and assertions
- Language interoperability.
- Composing threaded software.
- Composing concurrency.
Language Interoperability (aka “Babel 2.0”)  

- Current Babel features
  - Supports C, C++, Java, Fortran, Python
  - Provides common, portable object model
  - Allows clients in any supported language
  - Allows dynamic (re)composition of modules
    - Including during execution!
- Measured low overheads
  - Sometimes not low enough
- Carries “extras” with limited audiences
  - Contracts, RMI

- Planned COMPOSE-HPC improvements
  - Control via SIDL or source annotations
  - Optional use of object model
  - User-provided type mappings
  - User-controlled trade-offs between flexibility and optimization
    - Binary interoperability
    - Same-language short-circuits
    - Ability to inline calls
Composing Threads

- **Today**: entire applications are limited (at execution time) to a specific combination of MPI processes and threads per node
- **Tomorrow**: can we provide the flexibility to different modules within the application to use different “threading models”?  

Treat like node-local parallel remote method invocation (PRMI)

- Initially focus on control flow, handle data distribution manually

App: 2 MPI proc x 2 threads

PRMI adapters

Solver: 4 MPI proc

App: 2 MPI proc x 2 threads
Composing Concurrency

- **Today:** Most applications are SPMD
  - Looking for more parallelism at all levels
  - MPMD is hard, no tools
- **Tomorrow:** can we make MPMD programming routinely accessible?

- Remote procedure call (RPC) or remote method invocation (RMI) programming style
  - High-performance implementation
- Templates for common MPMD execution patterns
The Team

- **Galois, Inc.**
  - Matt Sottile

- **LLNL**
  - Tammy Dahlgren
  - Tom Epperly
  - Adrian Prantl

- **ORNL**
  - *David Bernholdt*
  - Wael Elwasif
  - Samantha Foley

- **PNNL**
  - Manoj Krishnan
  - Daniel Chavarria

- **SNL**
  - Ben Allan
  - *Rob Armstrong*
  - Geoff Hulette
Summary

- Flexible, evolvable, HPC code is hard to develop (and maintain).
- “Components” can help organize and develop large scale distributed components
  - As long as external interfaces are well defined and respected.
- Ability to
  - Specify aspects of your code at a high level, and
  - Auto-generate and integrate such aspects into the main code
Can be extremely beneficial (if done right).
- Source code annotation could be a convenient location to specify various aspects of HPC code
- Exascale could be a game changer for HPC codes
  - This maybe THE long awaited paradigm shift