SAND2008-3195C

The Acro/COLIN Framework: Developing Flexible Optimization Interfaces for Parallel, Hybrid, and Dynamically-Configured Algorithms

William E. Hart[†] and John D. Siirola[‡]

[†]Discrete Math and Complex Systems Dept.
[‡]Exploratory Simulation Technologies Dept.
Sandia National Laboratories
Albuquerque, NM USA

SIAM OP08 – 12 May 2008

{wehart, jdsiiro}@sandia.gov

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Motivation

- Simplify research and development of *hybrid algorithms*
  - "Traditional" branch-and-bound algorithms
  - "Traditional" sequential multi-solver hybrids
  - Asynchronous, adaptive multi-solver hybrids
  - Integrated exact and approximate (surrogate) models
  - Simultaneous optimization and uncertainty sampling
  - Multi-criteria optimization
  - Global optimization

- *This is more than just providing the result of one algorithm as the input to another.*

The challenge of (hybrid) optimization research

- What "we" talk about
  - Algorithm Development

- What we assume "just works"
  - Model evaluation
    - Residual calculation / external function calls
    - Sub-problem generation / formulation transformations
  - Data mgmt.
    - Type compatibility and transformation
    - Evaluation caching
    - Consistency mgmt.
  - Parallel resource mgmt.
  - Application integration
  - Configuration mgmt.

< 20% of the implementation effort?

> 80% of the implementation effort?
(more for parallel hybrid algorithms)

The promise of optimization frameworks

- Standardize access to common functionality, especially
  - function evaluations (residuals, Jacobian, Hessian)
  - solution management (storage & caching)
- Facilitate re-use of algorithmic components
  - e.g. line searches
- Provide common access points
  - e.g. for multi-solver hybridization; interfacing with a general sub-solver
- Enhance portability
  - hides platform-specific details
    - e.g. driving external executables
  - hides configuration-driven complexity
    - e.g. serial vs. threaded vs. distributed evaluation
- Assist V&V, SQE:
  - rigorously (and independently) test individual components
- *To make our lives easier.*

The limitation of “frameworks”

- Frameworks rely on a *rigid API*
  - Most written in common strongly-typed languages
    - C, C++, Fortran
  - API dictates:
    - concepts: organization (classes), functions, methods
    - semantics: parameters & data types
  - Requires “glue code”
    - “wrap” algorithm, function evaluation in framework classes
    - conversions to / from API data types
  - Application limited to API design space
    - e.g. if the API domain is {vector<int>, vector<double>}, you cannot accommodate complex numbers or S-expressions
    - API often offers gross superset of application’s needs
      - application must verify that correct subset is present
      - adding to API space requires modification of ALL clients

Flexible frameworks

- Optimization only requires a *conceptual framework*
  - Define organization and methods
    - "set domain state", "compute response", "store solution"
  - Actual data type (mostly) irrelevant to the framework
    - decouple interface methods from data types they operate on
  - Java's "everything's an Object" model is insufficient
    - method must expect and resolve explicit type caller provides
  - Scripting languages (i.e. Perl, Python) are closer
    - implicit conversion among scalar types (int, real, string, etc.)
    - implicit conversion of scalar to vector, hash to vector

COLIN: a Common Optimization Library INterface

- Object-oriented *conceptual framework*
  - Easily extensible C++ structure to wrap third-party solvers
    - add framework capabilities without modifying existing clients
    - clients may declare restricted framework interface
  - Facilitates plug-and-play optimizers for hybrid methods
  - Decouple component data types
    - e.g. solver uses `DakotaArray`, model uses `vector<double>`
  - Dynamic mappings
    - between problem domains
    - between problem types (e.g. LP, NLP, MINLP)
  - Support serial and parallel environments

Note: our goal is not to create the 'best' OO optimization class hierarchy … whatever that means ...

- Rather, it is to create a flexible and extendable framework that helps more than it hinders algorithm research.

COLIN 3.0

- Leverages concrete variant data type system
  - based on boost::any
  - three key components:
    - Any - concrete container for *any* data type / reference
    - TypeManager - casting system to convert Any into concrete types
    - Serializer - store & retrieve “Any”s without knowing type
- Interfaces specify inputs / outputs as “Any”s
  - concrete classes / methods
    - internally, extract parameters into concrete types
    - return concrete results wrapped in “Any”s
  - enables virtualization, dynamic linking
- Template traits to specify key framework properties & capabilities
  - solver & model types (LP, NLP, MINLP, etc.)

Core COLIN classes

- OptSolver
  - Generic optimizer base class
- OptProblem
  - Generic handle to an optimization problem
  - Templated on problem type
- OptApplication
  - Interface / driver for the actual model / function evaluator
  - Templated on problem type

- EvaluationManager
  - Manages serial / parallel / async evaluation requests
- BasicCache
  - Solution database
- TypeManager
  - Facilitates type conversions (data and problem types)

A Simple Example...

```
// A test function and its OptApplication driver
Any func(Any& point);
DirectAnyFuncApplication app = new
  DirectAnyFuncApplication<colin::NLP0_problem>(func));
app->configure_real_domain(3);
app->set_bounds("[-1.0,1.0]^3");

// Create and setup an optimization problem class
OptProblem<colin::NLP0_problem> prob;
prob.set_application(app);

// Create and setup a PatternSearch optimizer
PatternSearch opt;
opt.set_problem(prob);
opt.set_parameter("max_neval",100);

// Perform minimization and print the best value
opt.minimize();
cout << opt.best() << endl;
```

...hides complexity: automatic domain type mapping

```
// The actual test function
Any func(Any& point) {
   vector<double> domain;
   TypeManager().type_cast(point, domain);

   double ans = 0.0;
   for(size_t i=0; i<domain.size(); ++i)
      ans += domain[i]*somain[i];
   return ans;
}

// The PatternSearch optimizer
void PatternSearch::minimize() {
   array<utilib::Ereal> domain;
   utilib::Ereal response;

   // ...
   m_prob->EvalF(m_EvalMngr, domain, response);
   /* response type_cast called within EvalF() */
   // ...
}
```

Automatic problem reformulation

- Connect an non-gradient application to a gradient-based solver:

```
// Create and a derivative-free optimization problem
OptApplication<colin::NLP0_problem> *app;  // defined elsewhere
OptProblem<colin::NLP0_problem> prob;
prob.set_application(app);

// Create and setup a gradient-based optimizer (OPT++)
OPTpp opt;
opt.set_problem(prob);
```

`set_problem()` recognizes `prob` is not a `NLP1_problem` and leverages the TypeManager to convert `prob` into an `NLP1_problem` by reformulating it with a `FiniteDifferenceApplication`

Standard automatic reformulations

Name	Purpose	Examples
Downcast	Hide gradient or Hessian capability	NLP1 → NLP0 NLP2 → NLP1
Finite Difference	Add gradient capability	NLP0 → NLP1 MINLP0 → MINLP1
Constraint Penalty	Remove constraints and add penalty to objective	NLP1 → UNLP1
Subspace	Add or remove inactive portions of the domain	NLP1 → MINLP1 MINLP1 → NLP1[1]
Sampling	Create an approximate deterministic application by sampling a stochastic application	SNLP0 → NLP0

[1] - only succeeds if all integer variables are fixed

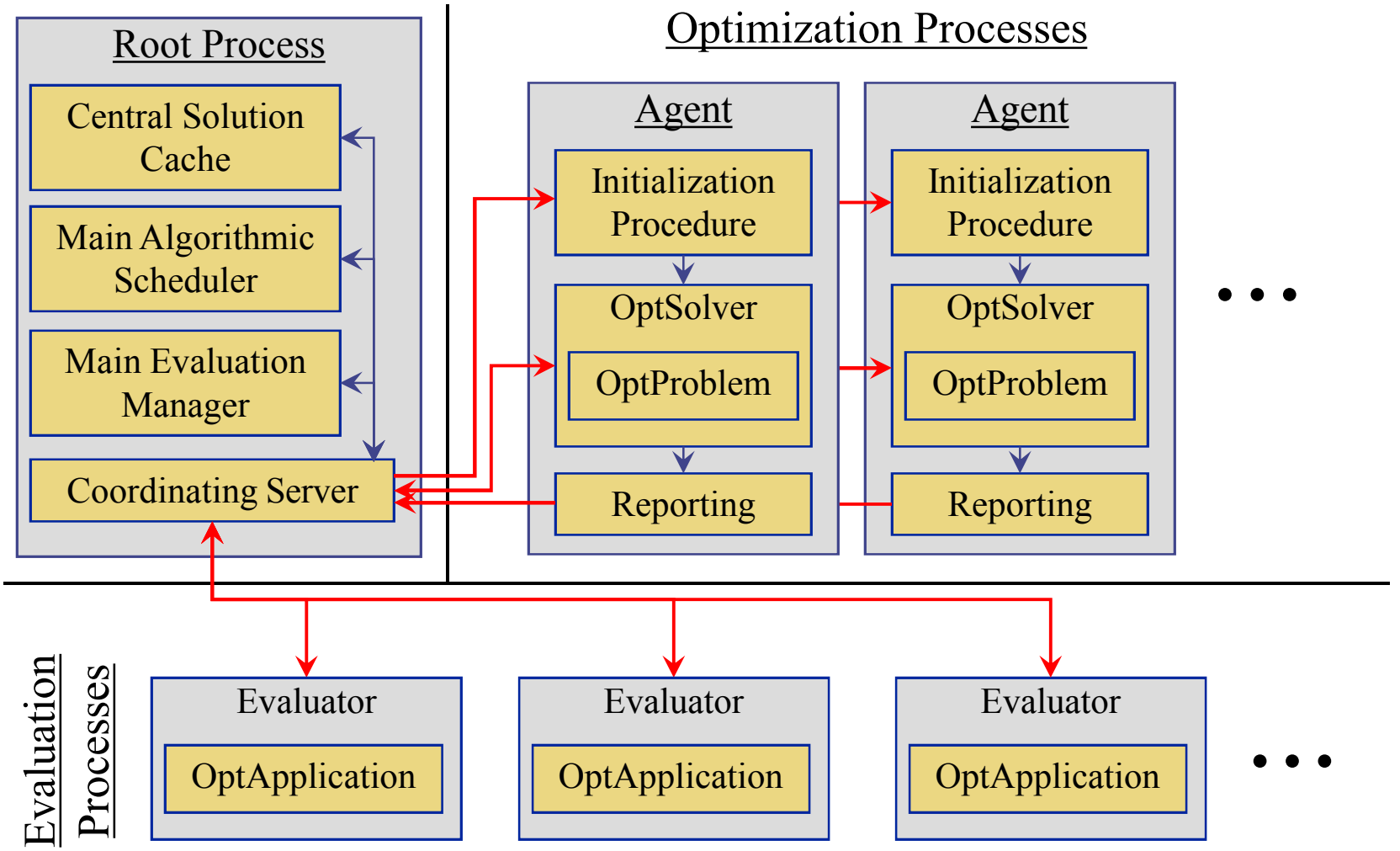
Acro: A Common Repository for Optimizers

- A central repository for optimization research
  - COLIN
  - Coliny – a collection of optimizers built on COLIN
    - Pattern Search
    - MultiState Pattern Search
    - Evolutionary Optimizers
  - Interfaces – wrappers for “COLINized” TPL algorithms
    - APPS
    - Cobyla
    - DOT
    - Hooke
    - MOMHLib
    - NPSOL
  - Agent-based Optimization

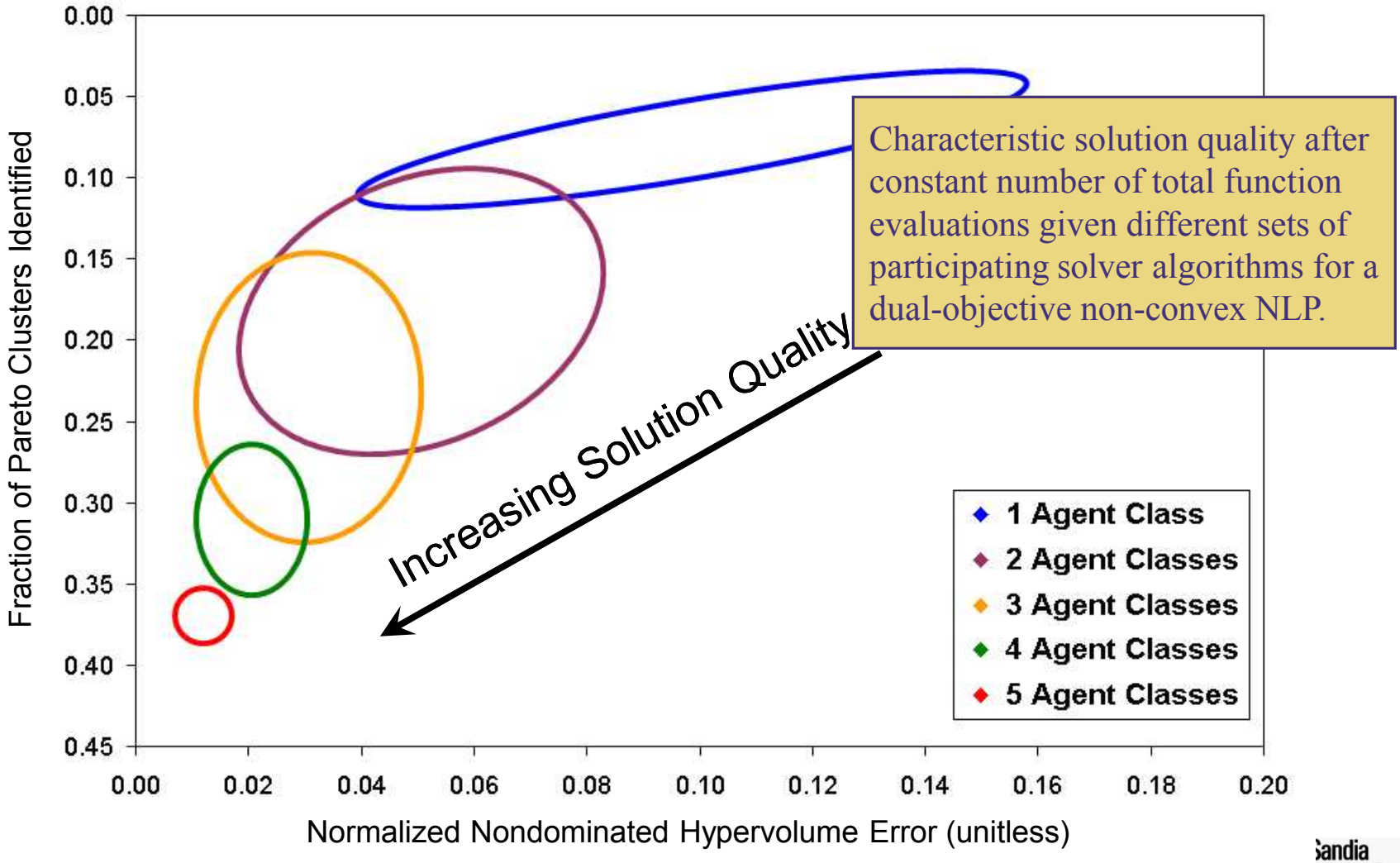
ABO: Agent-based Optimization

- Asynchronous, parallel, adaptive hybrid optimization
  - Reconciling a disparity in HPC optimization:
    - algorithms: serial; single-threaded; no silver bullet
    - architectures: distributed memory clusters
  - Instead of building the perfect parallel algorithm, run many algorithms in parallel
    - *Collaborative problem solving*
    - Algorithms → "agents"
      - independent entities that observe environment (candidate solutions)
      - react by applying algorithm and proposing new solutions
      - circular data flow: [Talukdar (1993)]
        - Initialize from Environment – Compute – Report to Environment
  - Target the "hardest of the hard" problems:
    - mixed-integer, nonlinear, nonconvex
    - multi-objective
    - wrought with uncertainty (may require sampling truth model)
    - "black box"

ABO Architecture



Example ABO study: effect of solver diversity



Characteristic solution quality after constant number of total function evaluations given different sets of participating solver algorithms for a dual-objective non-convex NLP.

[Siirola et al. *Comp Chem Eng*. 29(1) 2004 pp. 113-126]

Key enabling technologies

- Automatic domain translation
  - *Greatly* simplifies wrapping and including new algorithms
- Automatic problem translation / reformulation
  - Each solver receives the problem interface it "expects"
    - gradient-based local search → NLP1
    - pattern search → NLP0
    - stochastic search algorithms → UNLP0
    - ...etc.
  - Solvers only verify declared problem interface
    - guaranteed not to receive unexpected interface
- Variant-based infrastructure (utilib::Any)
  - Domain-independent solution caching
  - Serialization for inter-process communication
- Dynamic composition of agents:
  - OptSolver + Initialization + OptProblem

Future research enabled through COLIN

- Rapid development and testing of hybrid algorithms
  - "Traditional" hybrid or multi-solver algorithms
  - Parallel multi-solver algorithms
  - Distributed asynchronous multi-solver environments

- Dynamic formulation transformation
  - Dynamic generation of sub-problems
  - Dynamic problem reformulation to exploit solver properties
  - Multi-formulation hybrids

Software Availability

- Acro / COLIN:
  - Open Source
  - http://software.sandia.gov/Acro

- ABO:
  - general Open Source release targeted September, 2008.

COLIN 1.0 & 2.0

- C++ templates
  - interfaces templated on key framework data types
    - compiler will automatically coerce types
    - type incompatibilities discovered *at compile time*
  - limitations
    - numerous template instantiations lead to executable bloat
    - restricts use of virtual methods
    - limits use of dynamic (shared) libraries
    - requires many functions so compiler can coerce types
    - cannot handle truly *any* data type
      - template type's API set by use:

```
template<typename DOMAIN>
void reset_domain(DOMAIN d) {
  d.clear();
}
```

Forces the domain type to support a `clear()` method

Hybrid optimization frameworks