Are there Components in Auto-tuning?

Jeffrey K. Hollingsworth
University of Maryland
hollings@cs.umd.edu

Active Harmony

Automated Performance Tuning 101

- **Goal: Maximize achieved performance**
- **Problems:**
  - Large number of parameters to tune
  - Shape of objective function unknown
  - Multiple libraries and coupled applications
  - Analytical model may not be available
- **Requirements:**
  - Runtime tuning for long running programs
  - Don't try too many configurations
  - Avoid gradients

Active Harmony

- **Runtime performance optimization**
  - Can also support training runs
- **Automatic library selection (code)**
  - Monitor library performance
  - Switch library if necessary
- **Automatic performance tuning (parameter)**
  - Monitor system performance
  - Adjust runtime parameters
- **Hooks for Compiler Frameworks**
  - Working to integrate Utah & USC/ISI Chill

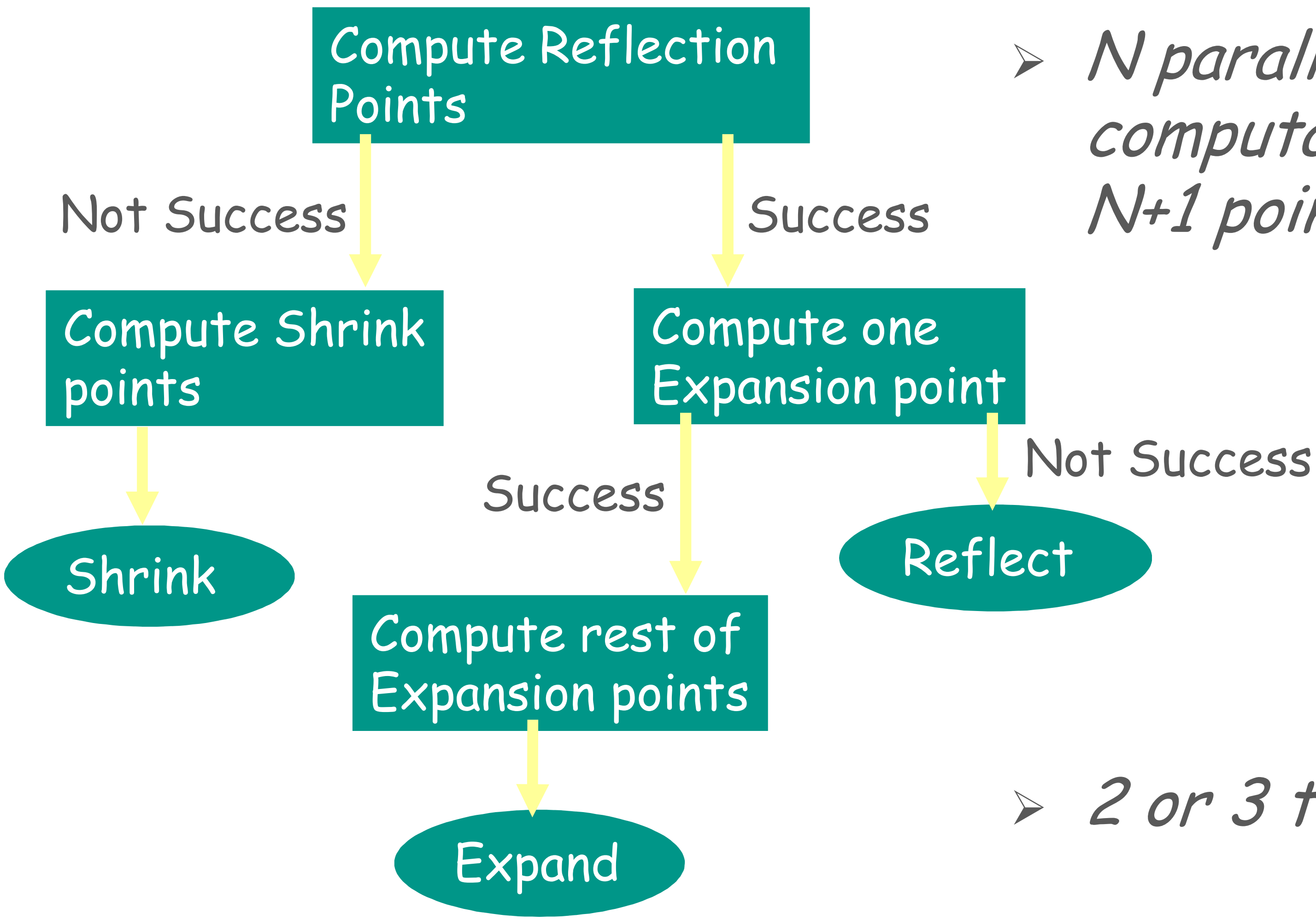
Possible Components

- **Making Auto-tuners plug into other tools**
- **Invoking External Search Point Instantiation**
  - Calls to generate a candidate configuration
- **Pluggable Search Algorithms**
- **Testing**
  - Programs to auto-tuning
  - Training objective functions

A Bit More About Harmony Search

- **Pre-execution**
  - Sensitivity Discovery Phase
  - Used to **Order** not **Eliminate** search dimensions
- **Online Algorithm**
  - Use Parallel Rank Order Search
    - Variation of Rank Oder Algorithm
      - Part of the class of Generating Set Algorithms
    - Different configurations on different nodes

Parallel Rank Ordering



- *N parallel computation for N+1 point simplex.*

- *2 or 3 time steps.*

But There Are Other Ways to Search

- **Different Algorithms**
  - Random
  - Hill Climbing
  - Simulated Annealing
  - Machine Learning Algorithms
  - …..

Component #1: Search API

- **Needed functionality**
  - Evaluate point
    - Run code at a point in search space
    - Likely to be a-sync to allow parallel search
  - Store/Read values for point in search space
    - Will include point in space, value, context (data set/machine info)
  - Query Spec
    - Learn about parameters, constraints
      - May use existing Math Prog API
    - Query Search Strategy Info

Search API

- **Related Questions**
  - Migrate ordering and grouping info to search API?
  - How can we use historical data?
    - Incorporating information from perf-db
  - Representation of the states
    - Types of iterators
    - “On Demand” evaluation needed to prevent space representation explosion

Component #2: Constraints

- **Define the search space:**
  - Represent the search space symbolically
  - Specify parameter types (integer vs. float)
  - Represent parameter domain (range, step etc.)
- **Represent constraints from:**
  - tools
  - applications (via automated analysis)
  - programmers
- **Provide support for arbitrary expression and function evaluation**

Requirements …

- **Express search hints:**
  - Ordering/ranking parameters (*unroll* before *tiling*)
  - Group parameters, code regions and/or constraints into sets
  - Represent data from static modeling, historical runs
- **Support for mapping language constructs**
  - Identify where in the source code (e.g. what loop) the optimization is taking place
- **Specify when and how to gather objective function value (compile-time vs. application launch-time)**

Specification Language

- **Six main components:**
  - Code Region Declaration
  - Region Set Declaration
  - Parameter Declaration
  - Constraint Declaration
  - Constraint Specification
  - Ordering Info
- **Provides a rich expression syntax**

Example Specification

```
parameter space simple_example
  {
        parameter x int {
            range [1:1:3];
                default 3;
            }

        parameter y int {
                range [1:1:3];
                default 2;
            }

        parameter z int {
                range [1:1:3];
                default 1;
            }

# And then the constraints.
            constraint c1 {
                    x≥z;
            }

            constraint c2 {
                    y>z;
             }

            # Constraint specification.
             specification  {
                    c1 AND c2;
             }

             # Ordering information is
                optional.
  }
```

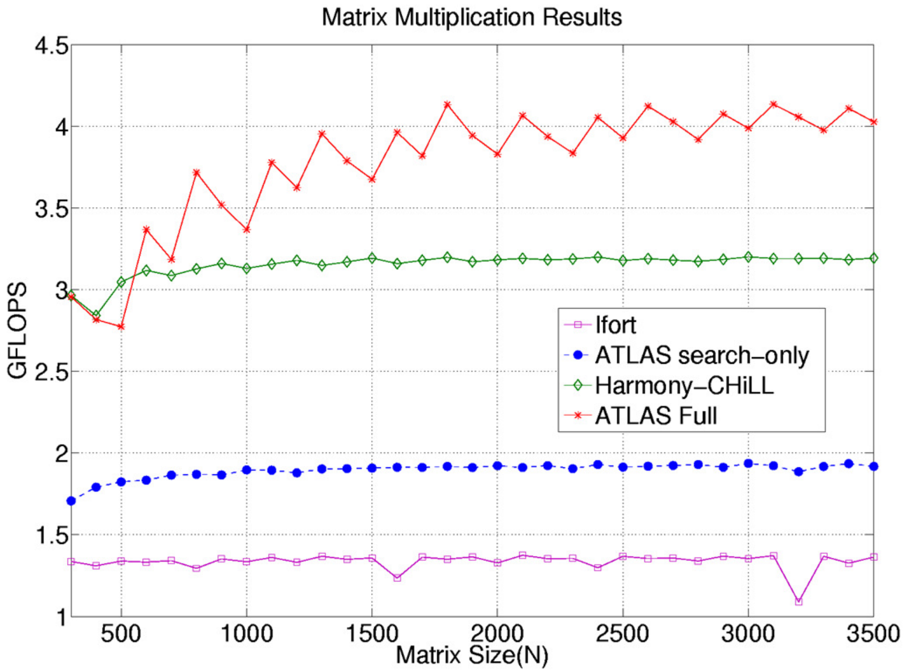
A Compiler Transformation Spec

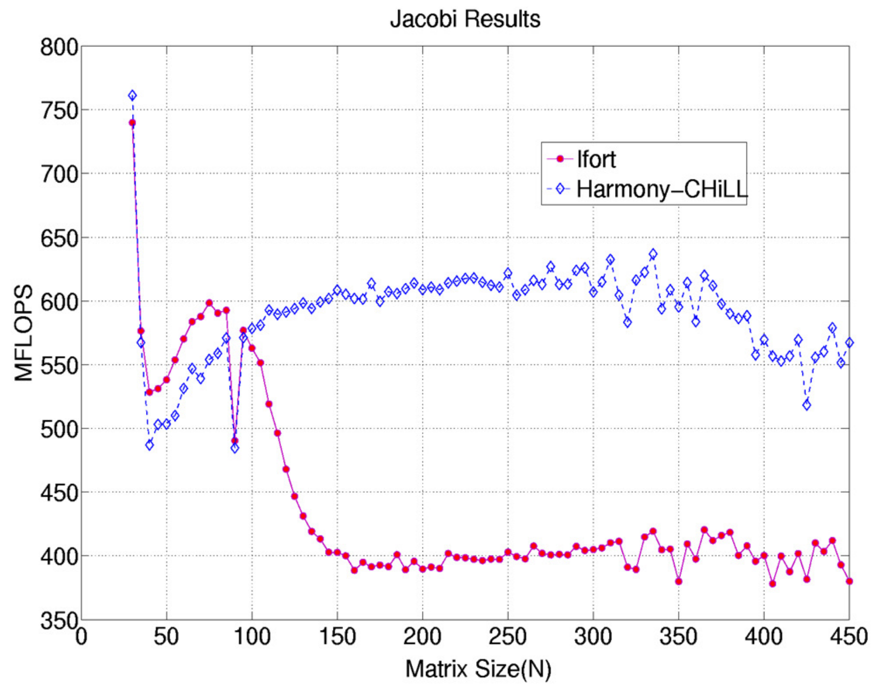
```
parameter space tiling {
   code_region loopI;
   code_region loopJ;
   region_set loop [loopI, loopJ];
   # declare tile_size parameter
   parameter tile_size int {
           range [2:2:256]
           default 32;
           region loop;
       }

   # Arbitrary constraint
   constraint c1 {
      (loopI.tile_size *
       loopJ.tile_size * 3 * 4) ≤
        2048;
   }

   # rectangular tiles better.
   constraint c2 {
      loopI.tile_size > loopJ.tile_size;
   }

   constraint c3 {
       loopJ.tile_size > loopI.tile_size;
    }

   specification {
       (c1 AND c2) OR (c1 AND c3);
    }
}
```

Component #3: Search Point Instantiation

- **Chill Compiler Transformations**
- **Described as a series of Recipes**
- **Recipes consist of a sequence of operations**
  - permute([stmt],order): change the loop order
  - tile(stmt,loop,size,[outer-loop]): tile loop at level loop
  - unroll(stmt,loop,size): unroll stmt's loop at level loop
  - datacopy(stmt,loop,array,[index]):
    - Make a local copy of the data
  - split(stmt,loop,condition): split stmt's loop level loop into multiple loops
  - nonsingular(matrix)

Tool Integration: CHiLL + Active Harmony





Generate and evaluate different optimizations that would have been prohibitively time consuming for a programmer to explore manually.

Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, Jeffrey K. Hollingsworth, “A Scalable Auto-tuning Framework for Compiler Optimization,” IPDPS 2009, Rome, May 2009.

SMG2000 Optimization

Outlined Code

```
for (si = 0; si < stencil_size; si++)
    for (kk = 0; kk < hypre__mz; kk++)
        for (jj = 0; jj < hypre__my; jj++)
            for (ii = 0; ii < hypre__mx; ii++)
                rp[((ri+ii)+(jj*hypre__sy3))+(kk*hypre__sz3)] -=
                    ((Ap_0[((ii+(jj*hypre__sy1))+ (kk*hypre__sz1))+
                     (((A->data_indices)[i])[si])])*
                     (xp_0[((ii+(jj*hypre__sy2))+(kk*hypre__sz2))+(( *dxp_s)[si])]));
```

CHiLL Transformation Recipe

permute([2,3,1,4])
tile(0,4,TI)
tile(0,3,TJ)
tile(0,3,TK)
unroll(0,6,US)
unroll(0,7,UI)

Constraints on Search

$0 \leq TI, TJ, TK \leq 122$
$0 \leq UI \leq 16$
$0 \leq US \leq 10$
compilers ∈ {gcc, icc}

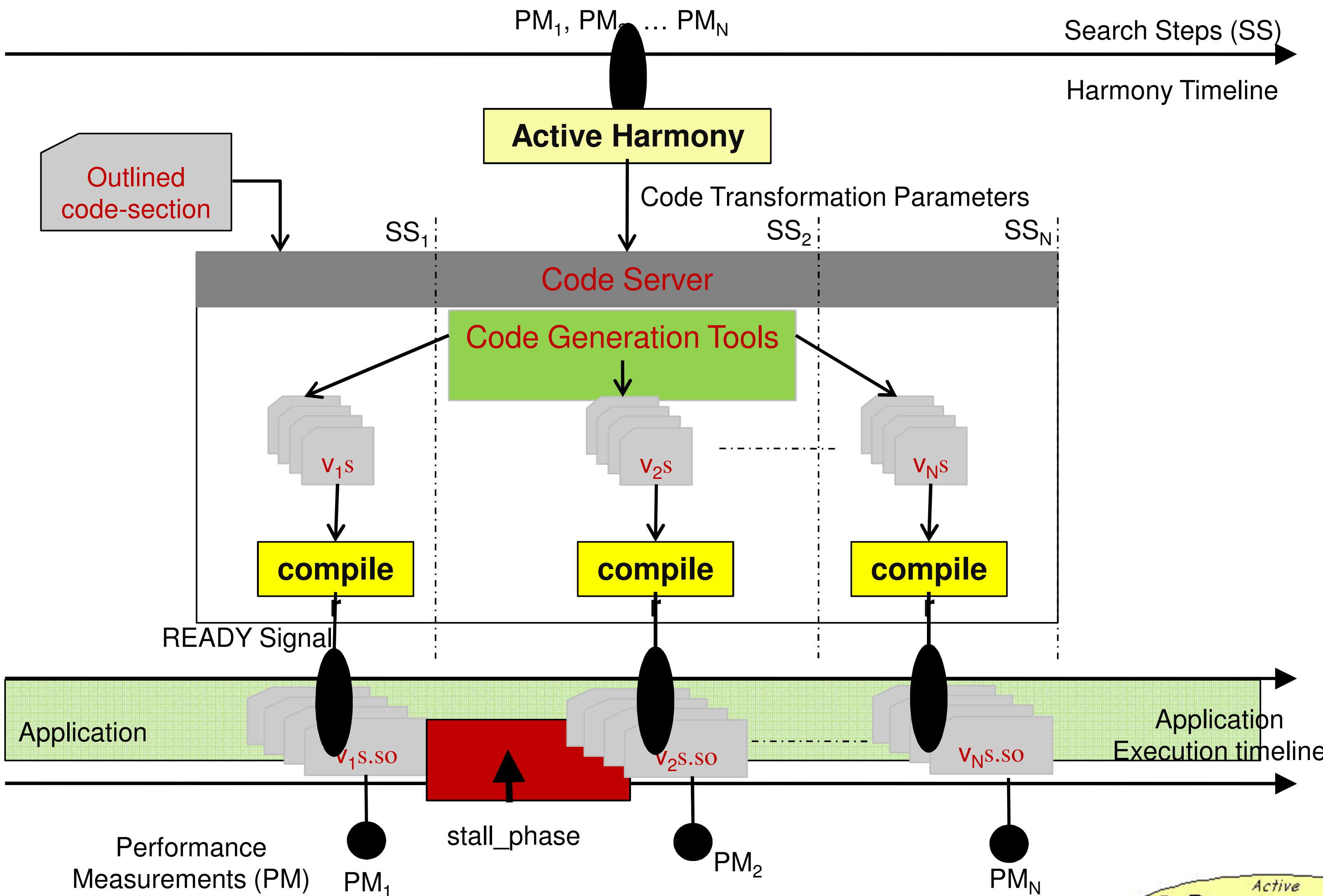
Search space:

122^3x16x10x2 = 581M points

Componentization Can Cause Changes

- **First level componentization**
  - Expose current functionality
  - Improve Testing
- **Second level**
  - Sometimes the next step requires internal changes
  - Adding new features to enable new uses

Compiling New Code Variants at Runtime



Online Code Generation Results

- **Three platforms**
  - umd-cluster (64 nodes, Intel Xeon dual-core nodes) – myrinet interconnect
  - Carver (1120 compute nodes, Intel Nehalem. two quad core processors) – infiniband interconnect
  - Hopper – (5,312 cores – two quad core processors, Cray XT5) – seaStar interconnect
- **Code servers**
  - UMD-cluster – local idle machines
  - Carver & Hopper – outsourced to a machine at umd
- **Codes**
  - PES - Poisson Solver (from Kelp distribution)
  - PMLB - Parallel Multi-block Lattice Boltzman

20

How Many Nodes to Generate Code?

- **Fixed parameters:**
  - Code: PES (poission solver)
  - problem-size (1024^3)
  - number of cores (128)
- **Up to 128 new variants are generated at each search step**

Code Servers	Search Steps+	Stalled steps+	Variations evaluated+	Speedup+
1	6*	46	502	0.75
2	17*	13	710	0.97
4	27	7.2	928	1.04
8	23	4.5	818	1.23
12	22	4.1	833	1.21
16	26	3.6	931	1.24

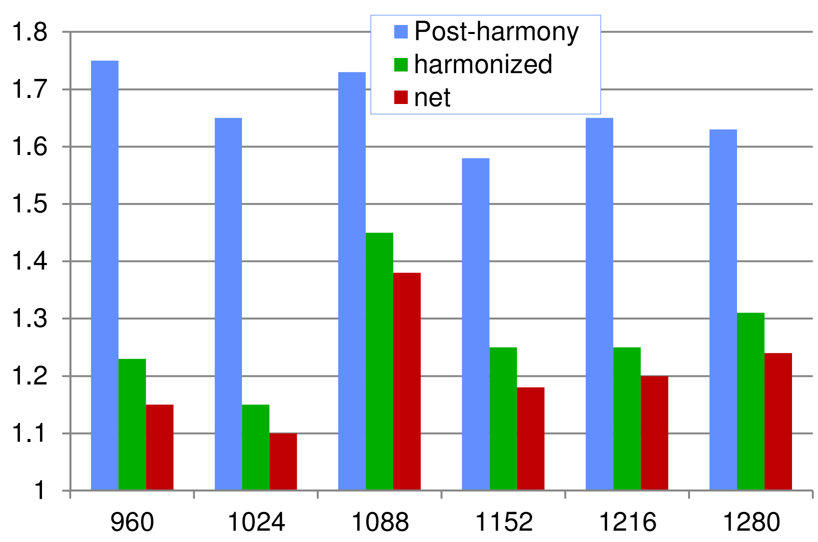
*** Search did not complete before application terminated**

+ Mean of 5 runs

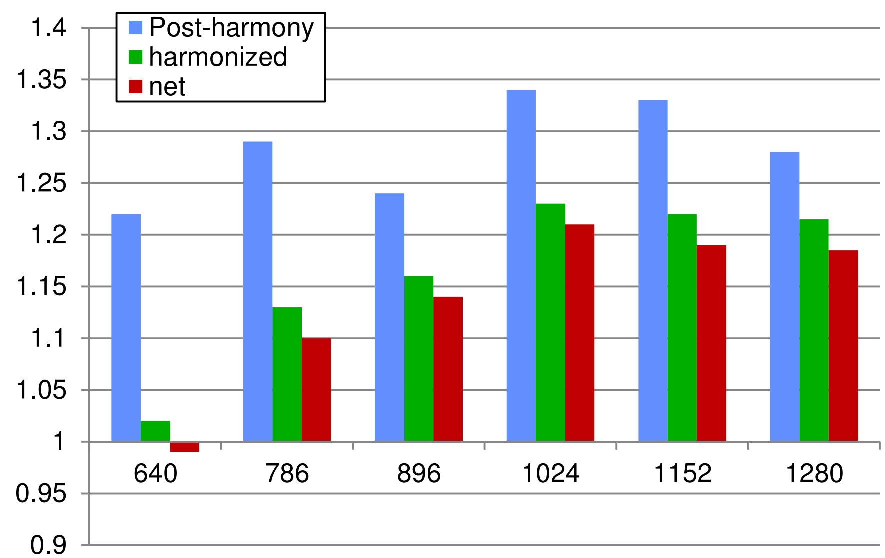
Runtime Code Generation Results

- All cases used 8 code servers
- Net is spedup factors in overhead of code generation cores
- Post-harmony is a second run using best config found in first
- X-axis is problem size

PES - 128 cores, UMD cluster



PMLB – 512 cores, Carver



Machine Specific Optimization

- Optimize for one machine, then run on others
- Results on speedups compared to base version
- Program is PES, all runs were 64 cores

	Run On UMD			Run On Carver			Run On Hopper		
Size	UMD	Carver	hopper	carver	UMD	hopper	Hopper	Carver	UMD
4483	1.42	1.13	1.00	1.51	1.38	1.34	1.28	1.30	1.27
5123	1.30	1.26	0.95	1.34	1.31	1.33	1.34	1.31	1.28
5763	1.38	1.16	1.02	1.42	1.39	1.27	1.31	1.35	1.30

Active Harmony

Component #3: CBTF + Harmony

- **Make Active Harmony a component in CBTF**
  - Consumer of performance Data
    - Uses other components to guide search
  - Supplier of performance Tuning
    - Results of experiments can be improved programs in addition to data
  - User of scalable control and collection system
    - Need to gather performance data from nodes
    - Send out changes to application and runtime
  - User of GUI and visualizations
    - We are not GUI experts
    - Uniform look and feel possible with CBTF

Active Harmony

Component #4: Test Data

- **Create a library of auto-tuning performance curves**
  - Include data points and objective values
  - Include multiple samples per point
  - Includes meta data
- **Precedence**
  - It's really just a benchmark of sorts
  - Optimization community has challenge datasets

Evaluating Componentization

- **Cleaner, more testable code**
- **Third part plugins appear**
- **Others start to use/add your components**
- **New ideas inspired by features**

Conclusions

- **Auto tuning Works!**
  - Real programs run faster
- **Component opportunities abound**
  - Between "competing" auto-tuning systems
  - As part of other component frameworks
- **Bonus benefits of components**
  - Better testing
  - Cleaned up code