Recent Results from PERI Autotuning

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

Protonu Basu (Utah), Jacqueline Chame (ISI), Chun Chen (Utah),
Rob Fowler (RENCI), Mary Hall (Utah), Kumar Mahinthakumar (NCSC),
Gabriel Marin **(Rice),** ***John Mellor-Crummy (Rice),***
Dan Quinlin (LLNL), Shreyas Ramlingam (Utah),
Vamsi Sripathi (NCSC), Jeffrey Tilson (RENCI), Ananta Tiwari (UMD)

SciDAC
Scientific Discovery through Advanced Computing

PERI

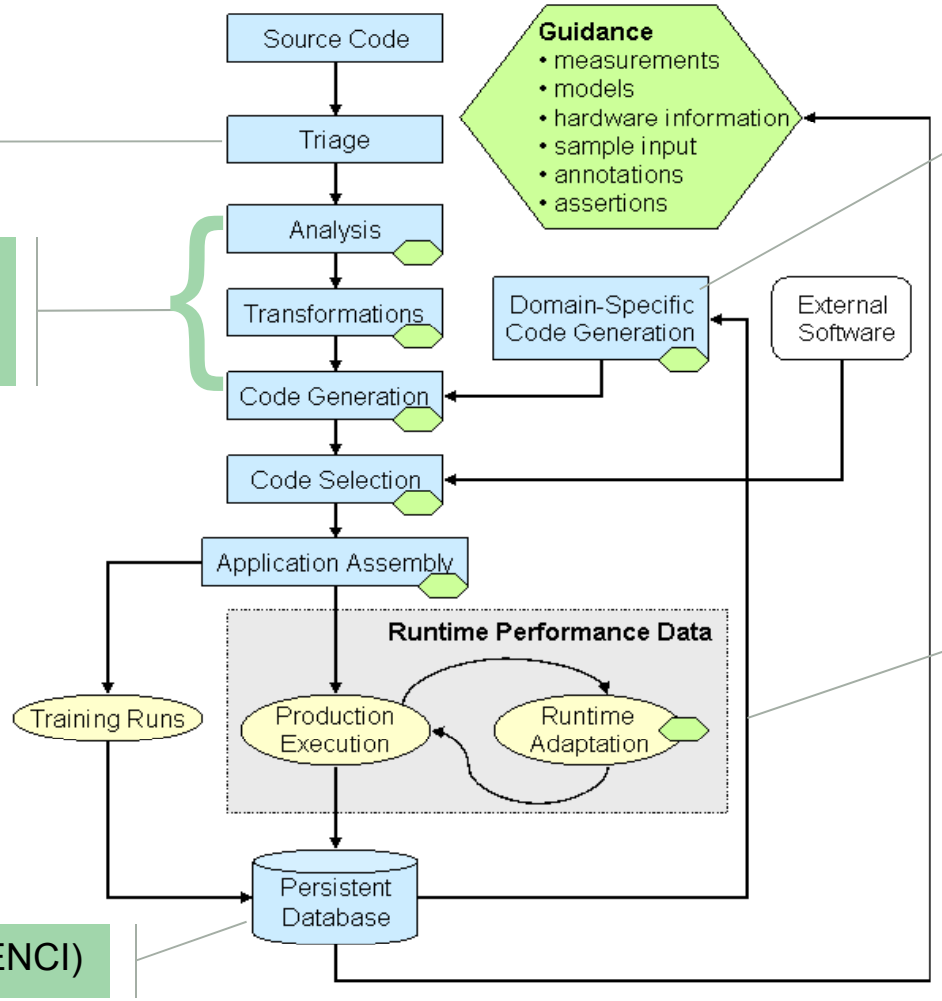
PERI Autotuning Vision

- **Long-term goal:**
  - End-to-end autotuning of key computations in DOE applications
  - Apply technology to whole programs at scale
- **Key ideas:**
  - Triage from source code to focus on key computation
  - Compiler and library technology to generate parameterized code variants and models
  - Gather empirical data of code variant performance
  - Search frameworks for feasible exploration of large space of possible implementations

PERI Autotuning Tools

HPC Toolkit (Rice)
ROSE (LLNL)

CHiLL (ISI and Utah)
ROSE (LLNL)
Orio (Argonne)



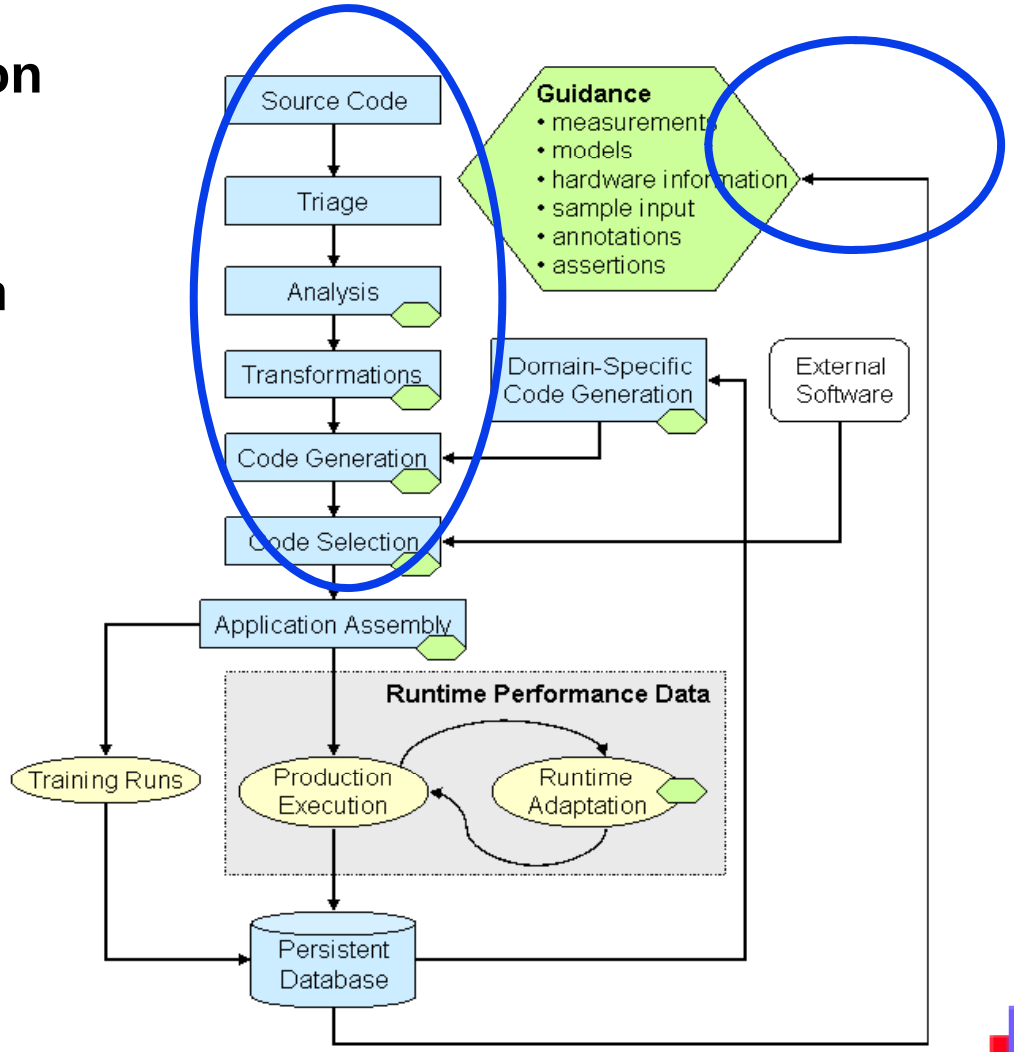
OSKI (LBNL)

Active Harmony (UMD)
GCO (UTK)

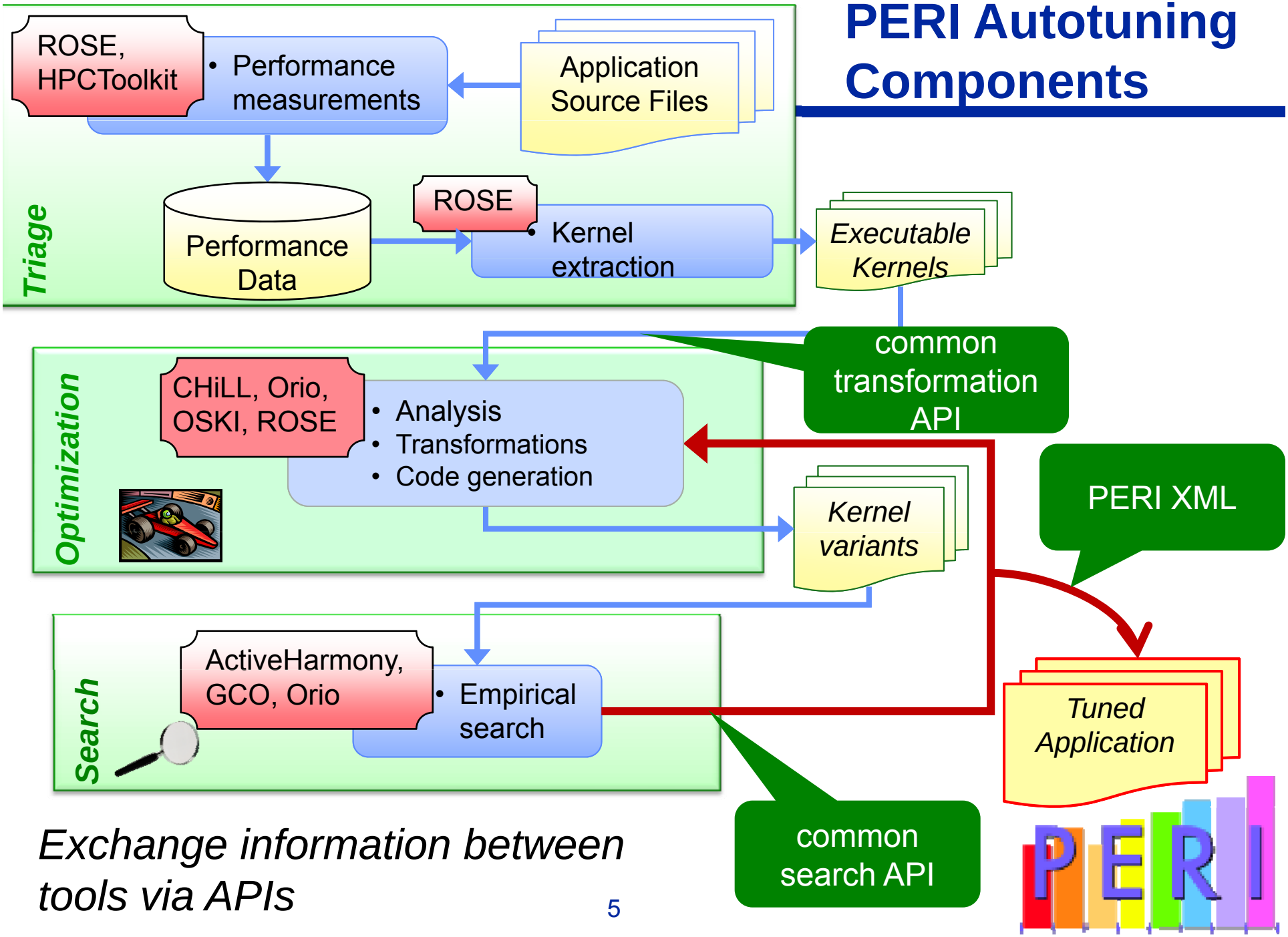
PerfTrack (LBNL, SDSC, RENCI)

Autotuning Tool Capabilities

- **Talk will focus on 3 Aspects:**
  - Triage
  - Optimization
  - Search



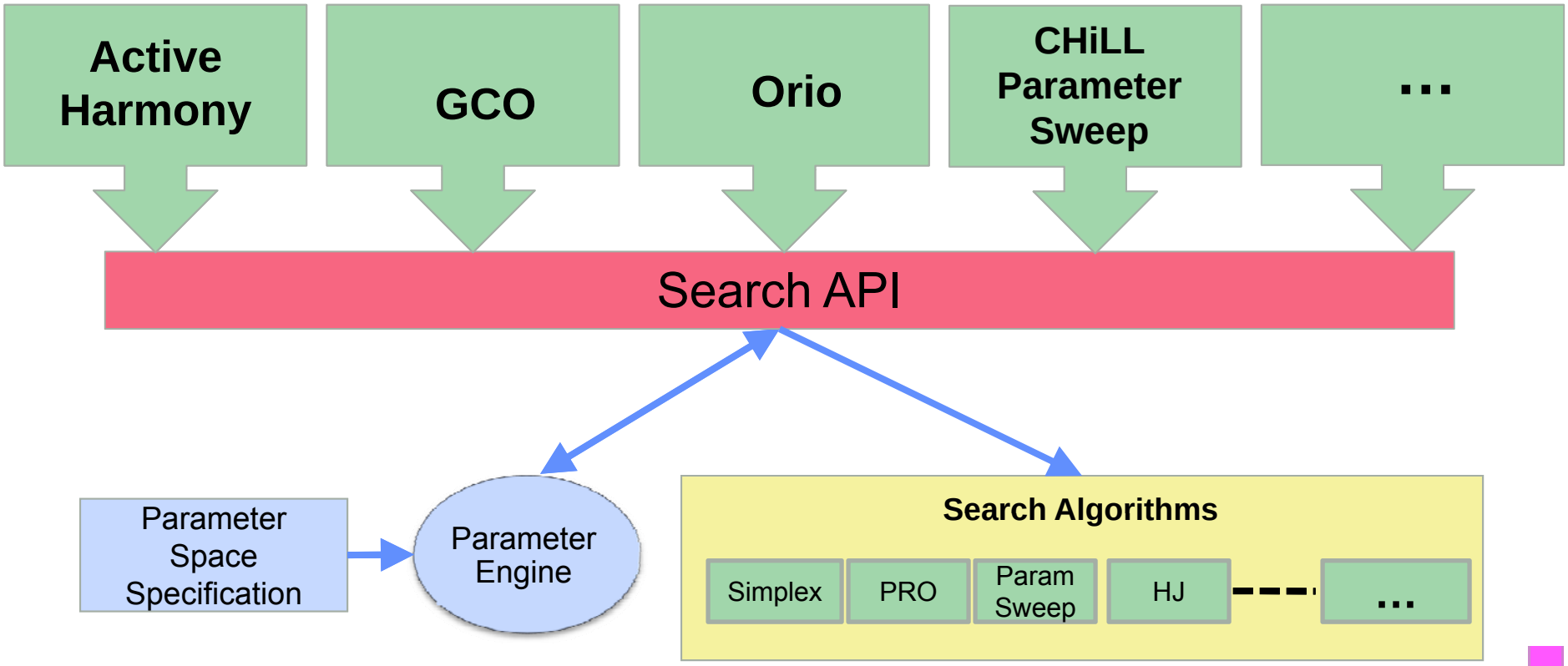
PERI Autotuning Components



Exchange information between tools via APIs

PERI Search API

- Search algorithms can be plugged into generalized search framework



Autotuning Tool Capabilities: Triage

- **Triage**
  - Focus in on key computations
  - Separate these from rest of the application
  - Hand off key computations to autotuning optimization process
- **PERI Tools**
  - HPCToolkit (Rice):
    - Sampling-based performance analysis of parallel applications
    - Maps these measurements back to code constructs
  - ROSE (LLNL):
    - Read performance information derived by HPCToolkit
    - Identify key computation kernels
    - Derive “outlined” standalone version of key computation kernels

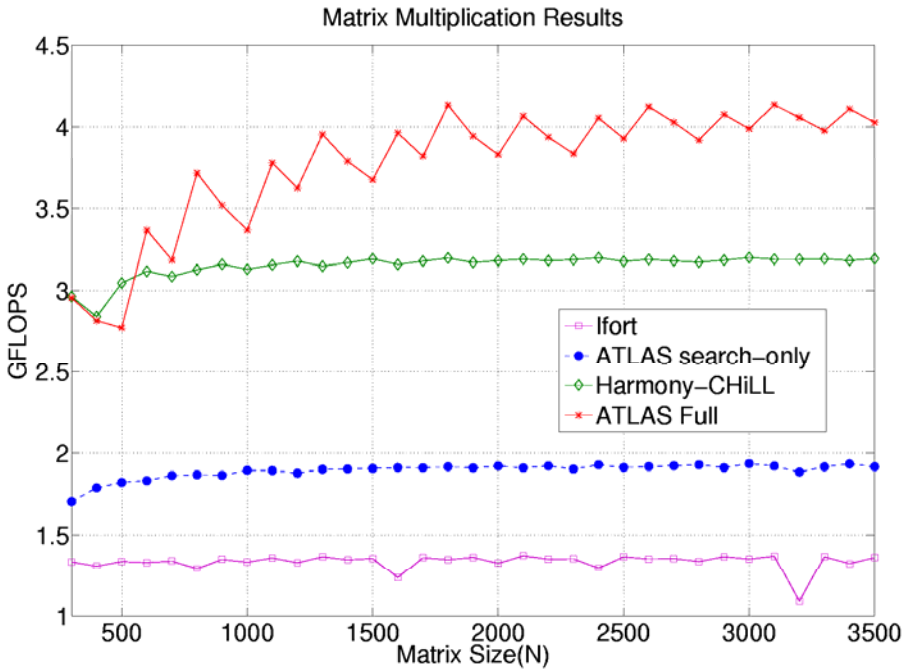
Autotuning Tool Capabilities: Optimization

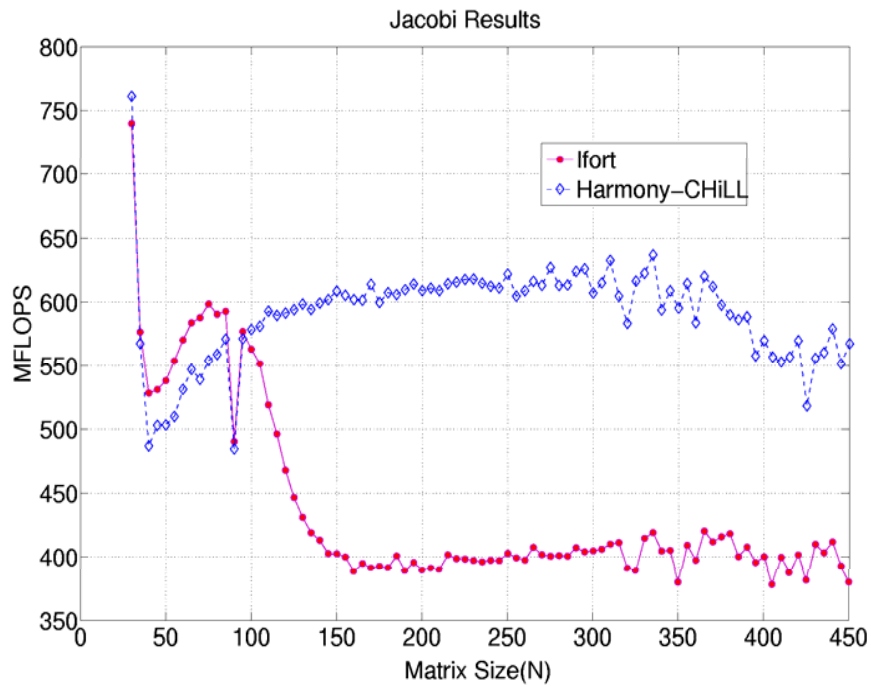
- **Optimization**
  - **Analysis:** Discover properties of kernel that suggest appropriate transformation (rewriting) strategies
  - **Code Transformation:** Evaluate alternative rewriting strategies and derive a set of transformation recipes
  - **Code Instantiation:** Generate code from transformation recipe
- **PERI Tools**
  - CHiLL (USC/ISI, Utah and Argonne):
    - Polyhedral analysis, transformation and code generation framework
  - ROSE (LLNL):
    - Analysis and POET code transformation from recipe
  - Orio (ANL):
    - Autotuning using annotations and high-level specification
  - OSKI (LBNL):
    - Self-tuning sparse matrix library

Autotuning Tool Capabilities: Search

- **Search**
  - Empirical evaluation of alternative mappings for key computations
  - Execute code with representative input, select next search point(s), repeat until converges
- **PERI Tools**
  - Active Harmony (UMD):
    - Parallel search of parameters and variants using the Parallel Rank Ordering Algorithm
  - GCO (UTK):
    - Suite of search techniques (PSO, GA, Simplex, Simulated Annealing, Orthogonal, Random, Exhaustive, and Hybrid) integrated with autotuning code generation

Tool Integration: CHiLL + Active Harmony

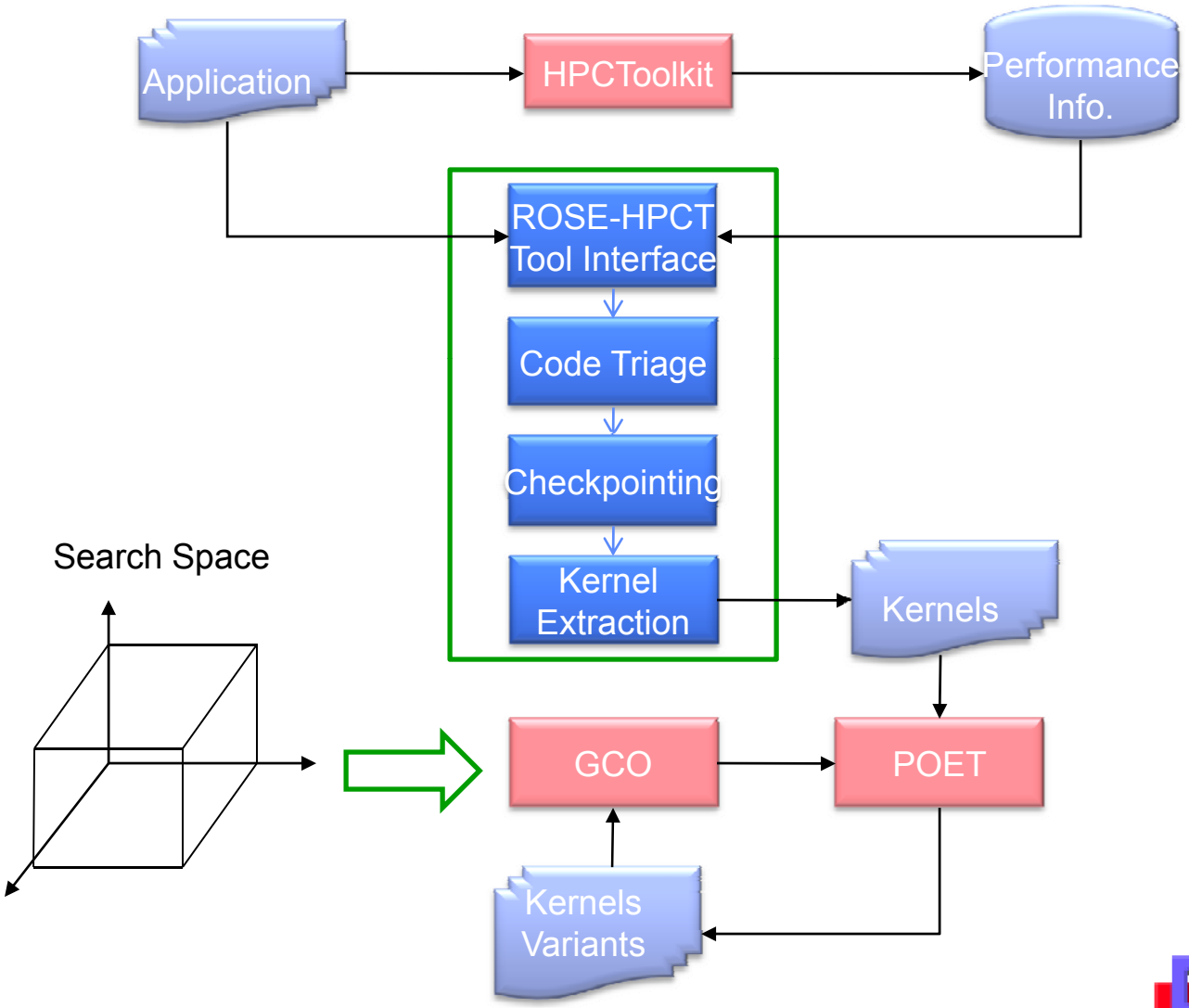




Generate and evaluate different optimizations that would have been prohibitively time consuming for a programmer to explore manually.

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

Integration: HPCToolkit + ROSE + POET + GCO

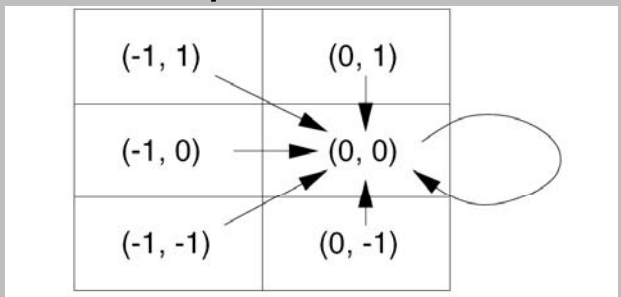


SMG2000 Case Study

- **Semi-coarsening multigrid on structured grids**
  - Residual computation contains sparse matrix-vector multiply bottleneck, expressed in 4-deep loop nest
  - Key computation identified by HPCToolkit

```
for si = 0 to NS-1
  for k = 0 to NZ-1
    for j = 0 to NY-1
      for i = 0 to NX-1
        r[i + j*JR + k*KR] -=
            A[i + j*JA + k*KA + SA[si]]
          * x[i + j*JX + k*KX + Sx[si]]
```

2D 6-point Stencil



$S = \{(-1,1),(-1,0),(-1,-1),(0,1),(0,0),(0,-1)\}$

SMG2000 Triage

HPCToolkit

hpcviewer:

File Debug Search Help

cyclic_reduction.c | smg_residual.c

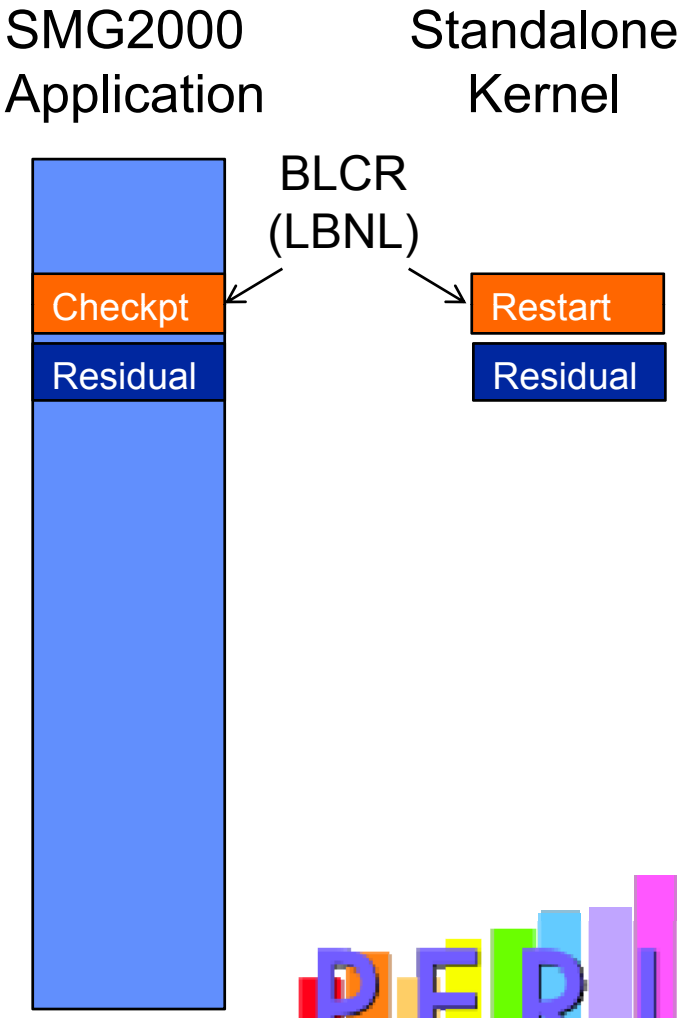
```
276                     xp = hypre_StructVectorBoxData(x, i) +
277                        hypre_BoxOffsetDistance(x_data_box, stencil_shape[si]);
278
279                     hypre_BoxGetStrideSize(compute_box, base_stride,
280                                            loop_size);
281                     hypre_BoxLoop3Begin(loop_size,
282                                         A_data_box, start, base_stride, Ai,
283                                         x_data_box, start, base_stride, xi,
284                                         r_data_box, start, base_stride, ri);
285#define HYPRE_BOX_SMP_PRIVATE loopk,loopi,loopj,Ai,xi,ri
286#include "hypre_box_smp_forloop.h"
287                     hypre_BoxLoop3For(loopi, loopj, loopk, Ai, xi, ri)
288                        {
289                           rp[ri] -= Ap[Ai] * xp[xi];
290                        }
291                     hypre_BoxLoop3End(Ai, xi, ri);
292                  }
293               }
294            }
295      }
296
297      /*--------------------------------------------------------------------
298       * Return
299       *--------------------------------------------------------------------*/
```

46% of execution time

Flat View

Scope	WALLCLK
Experiment Aggregate Metrics	7.28e04 100 %
Load module /home/liao/svnrepos/benchmarks/smg2000/test/smg2000	7.28e04 99.9%
smg_residual.c	4.22e04 58.0%
hypre_SMGResidual	4.22e04 58.0%
smg_residual.c: 289	3.38e04 46.5%
smg_residual.c: 152	6.20e03 8.5%
smg_residual.c: 287	8.04e02 1.1%
smg_residual.c: 236	7.12e02 1.0%
smg_residual.c: 238	5.66e02 0.8%

ROSE Outliner



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

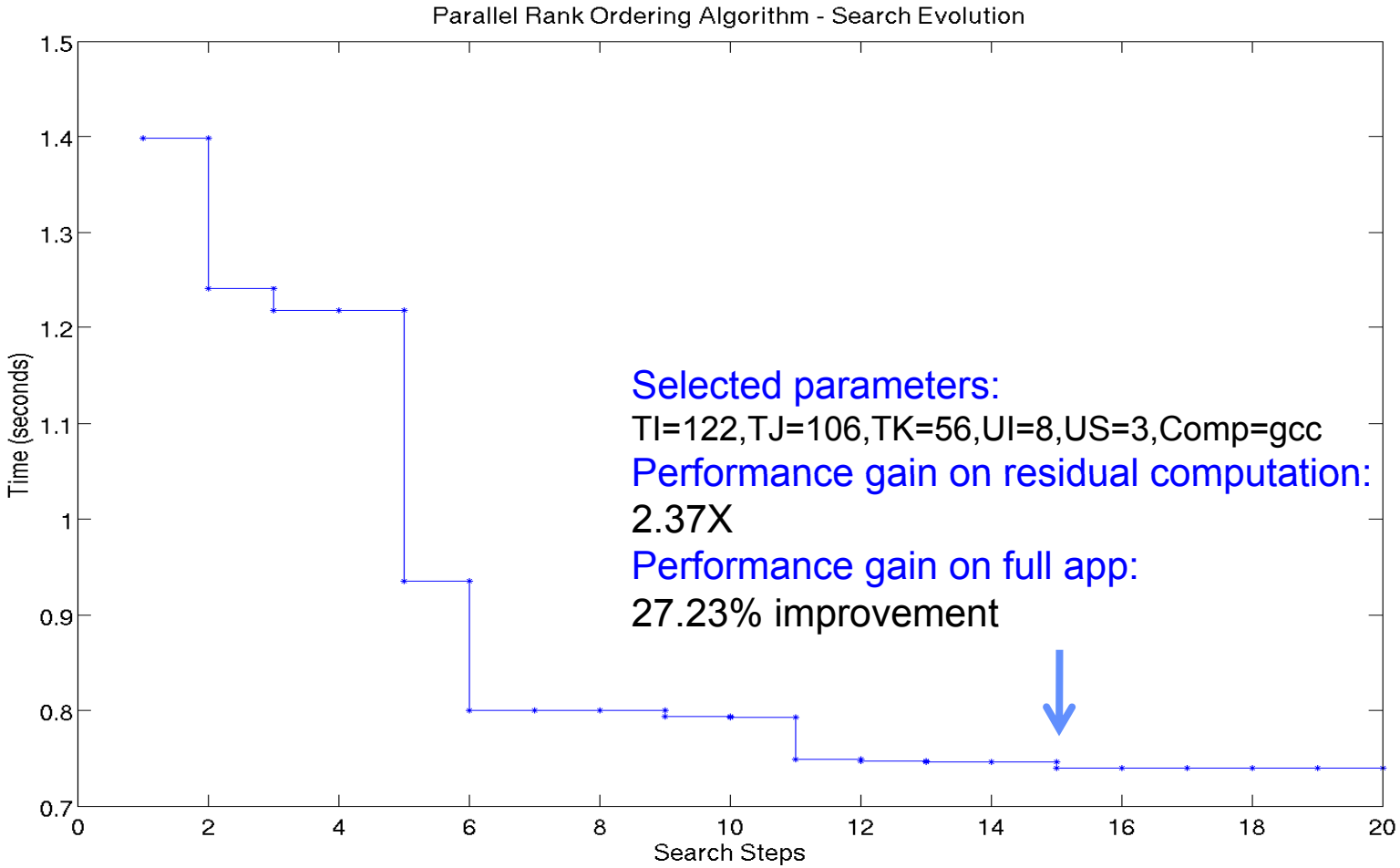
$0 \le TI, TJ, TK \le 122$
$0 \le UI \le 16$
$0 \le US \le 10$
compilers $\in$ {gcc, icc}

Search space:

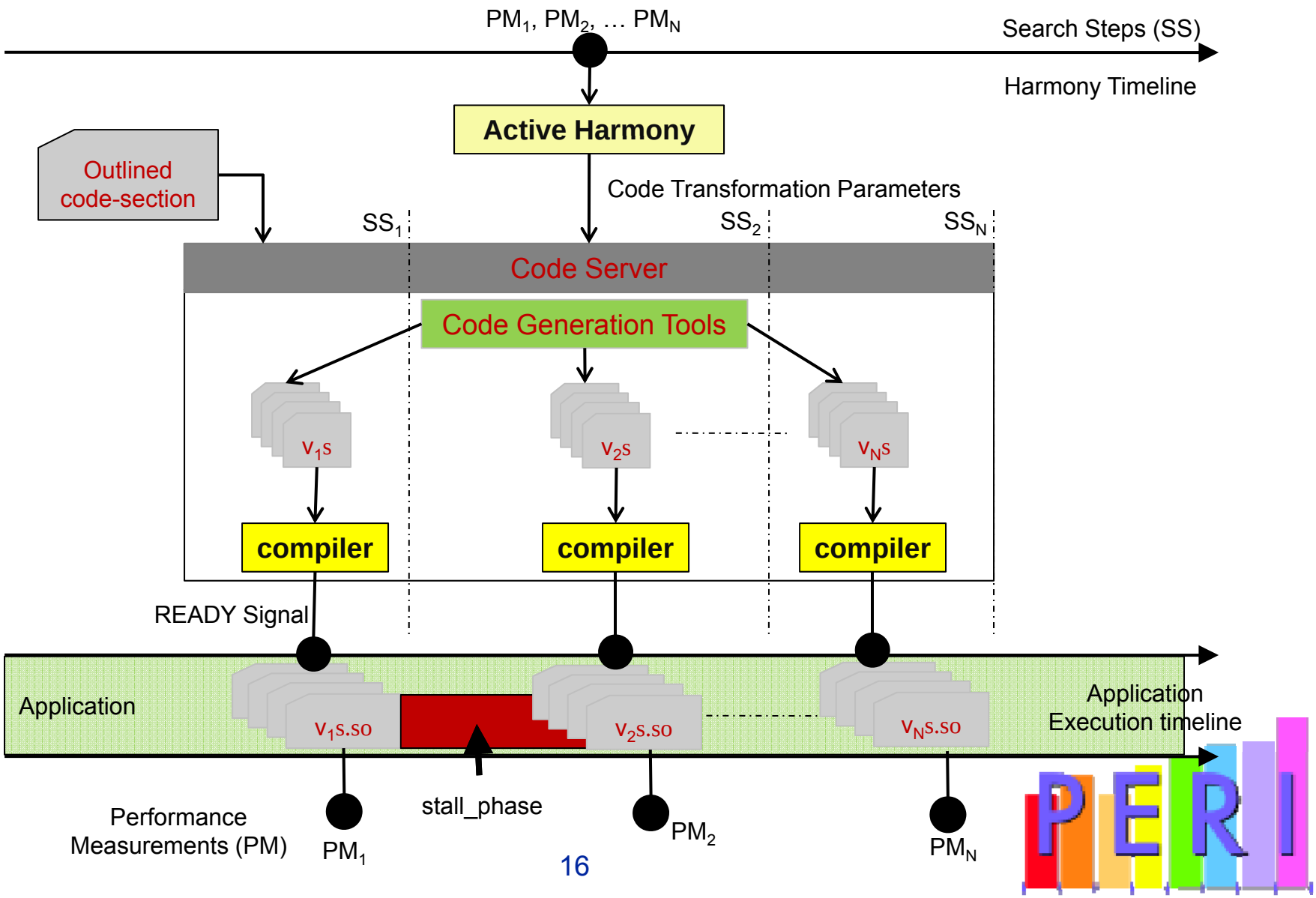
$122^3 \times 16 \times 10 \times 2 = 581M$ points

SMG2000 Search and Results

Parallel search evaluates 490 points and converges in 20 steps



Compiling New Code Variants at Runtime



Online Code Generation Results

- **Two platforms**
  - umd-cluster (64 nodes, Intel Xeon dual-core nodes) – myrinet interconnect
  - Carver (1120 compute nodes, Intel Nehalem. two quad core processors) – infiniband interconnect
- **Code servers**
  - UMD-cluster – local idle machines
  - Carver – outsourced to a machine at umd
- **Codes**
  - Poisson Solver
  - PMLB Parallel Multi-block Lattice Boltzman
  - SMG2000

How Many Nodes to Generate Code?

- **Fixed parameters:**
  - Code: poission solver
  - problem-size (1024^3)
  - number of processors (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

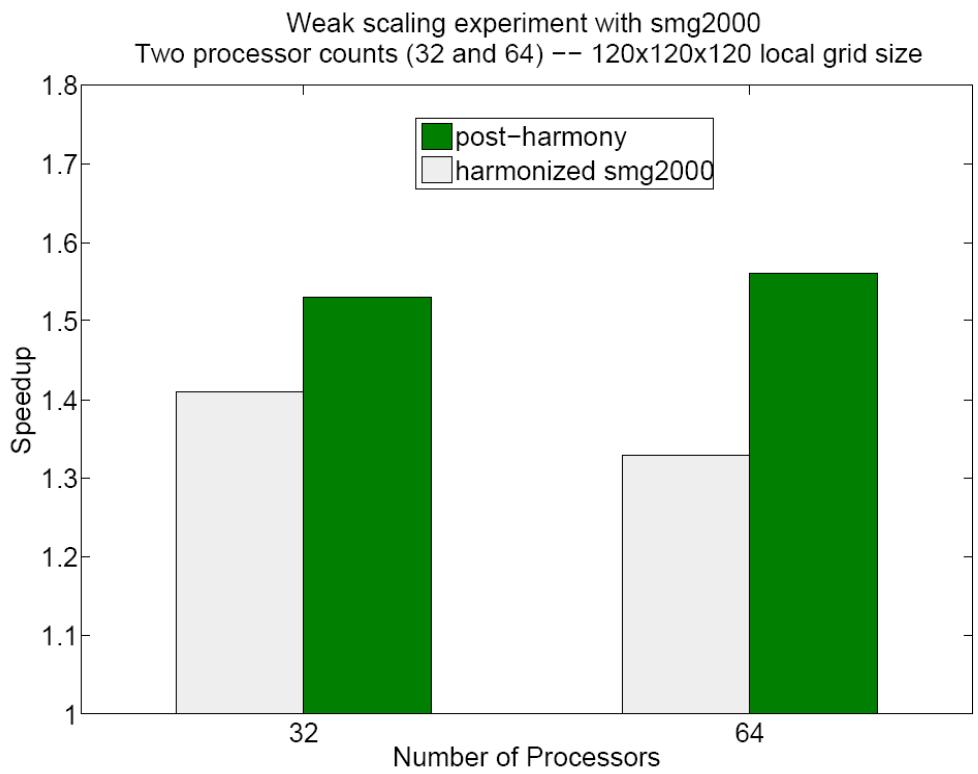
Auto Tuning For Different Platforms

- **Fixed parameters:**
  - Code: PMLB
  - Processors: 64
- **Study how parameters differ for the two systems**
- **Use harmony determined parameters from one system**
  - Run a post-line (fix parameters for entire run) run on another

Problem Size	Speedup (post-line) run on UMD Cluster		Speedup (post-line) run on Carver Cluster	
	UMD Best Config	Carver Best Config	Carver Best Config	UMD Best Config
384^3	1.44	1.19	1.32	1.30
448^3	1.42	1.13	1.51	1.38
512^3	1.30	1.26	1.34	1.30
576^3	1.38	1.16	1.42	1.39

SMG2000 Online Tuning Results

- **Online tuning for smg_residual function**
  - Tiling and unrolling
  - Up to 1.4x speedup within a single run



Autotuning compile and run command line options

Tuneable Parameters

- Application & Compiler options
- Linkage options
  - choice of libraries
  - static/dynamic
- Runtime flags
  - number of nodes, cores, ...

Applications

- GAMESS
  - Chemistry application
  - High complexity algorithms for solving quantum chemistry models
- ADCIRC/SWAN
  - Coupled multiscale/multiphysics system
  - Wave-storm surge coupling

Autotuning Flags Results

- **GAMESS: A quantum chemistry code (Linux cluster)**
  - Demonstrated the system on a well optimized code
  - Varied compiler options, # of nodes, and networking type (IB vs gigE)
  - Little sensitivity to most compiler options.
  - Demonstrated/Expected significant benefits to using IB over gigE for large numbers of nodes.
- **ADCIRC/SWAN: Coastal circulation & near shore wave models (BG/L)**
  - Examine many compiler options, with(out) libmass
  - TRied several compiler parameters
    - » -q(hot,O,arch,unwind,tune,cache) etc
  - 18% improvement in SWAN speed with non-obvious set of compiler options
  - Identified inaccurate documentation of compiler options
    - libmass was NOT automatically included for –O4.

PFloTran

- **PFloTran models Multiscale-Multiphase-Multicomponent Subsurface Reactive Flows**
- **Auto Tuning Approach**
  - Used profiling to identify critical routines:
    - PFloTran routine that computes the total component concentrations) **rtotal**
      - 7.9% of time, 4.5% of peak+
    - PETSc kernel: forward triangular solve routine (trisolve)
      - 9.8% of time, 4.0% of peak+
  - Outline routines and autotuned them
  - Autotuning consists of:
    - Chill compiler transformations
    - Changing compilers

+ run using 4k cores on Cray XT5 (Jaguar)

RTtotal Results

- **Tuning Methods**
  - Chill (no Harmony since search space is small)
    - Loop unrolling, specialization
  - Hand Tuned Code
    - Loop unrolling, specialization plus scalar replacements

Ncomp	Original	Chill		Hand Tuned	
	Time	Time	Speedup	Time	Speedup
1	0.09	0.06	1.52	0.05	1.80
2	0.13	0.096	1.32	0.087	1.46
3	0.18	0.12	1.49	0.11	1.45
4	0.21	0.16	1.32	0.15	1.45
5	0.27	0.20	1.36	0.18	1.53
6	0.32	0.27	1.22	0.23	1.42
7	0.39	0.32	1.25	0.27	1.48
8	0.43	0.38	1.11	0.30	1.42

Autotuning Trisolve

Outlined Code

```
#define SIZE 15
void forward_solve_kernel( … )  {
   ….
   for (cntr = SIZE - 1; cntr >= 0; cntr--) {
         x[cntr] = t + bs * (*vi ++);
         for (j=0; j<bs; j++)
             for (k=0; k<bs; k++)
                 s[k]-= v[cntr][bs* j+k] * x[cntr][j];
    }
}
```

CHiLL Transformation Recipe

```
original()
known(bs > 14)
known(bs < 16)
unroll(1,2,u1)
unroll(1,3,u2)
```

Constraints on Search

0 <= u1 <= 16

0 <= u2 <= 16

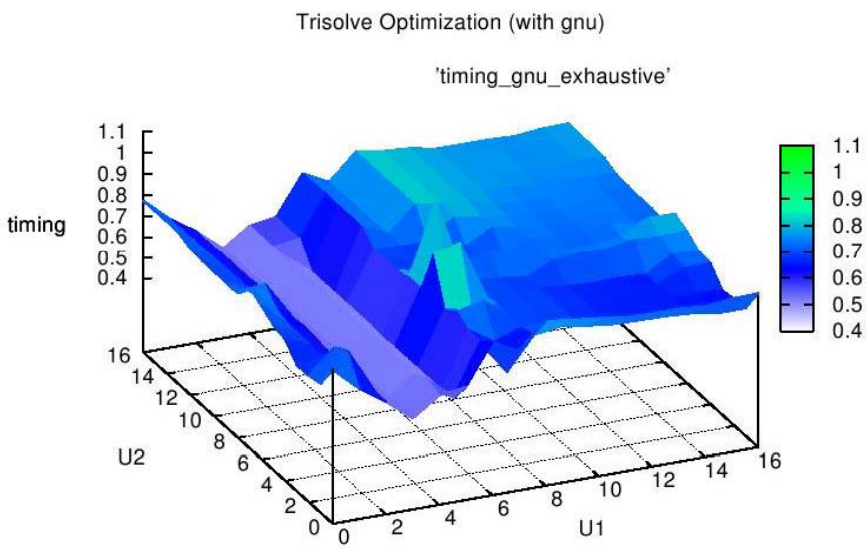
compilers ∈ {gnu, pathscale, cray, pgi}

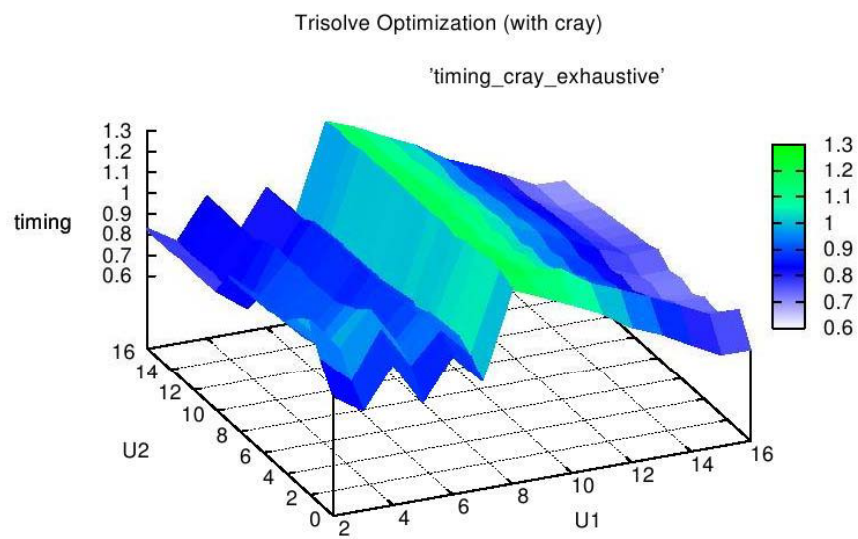
Search space:

17x17x4 = 1156 points

Trisolve Results

Compiler	Original	Active Harmony			Exhaustive		
	Time	Time	(u1,u2)	Speedup	Time	(u1,u2)	Speedup
pathscale	0.58	0.32	(3,11)	1.81	0.30	(3,15)	1.93
gnu	0.71	0.47	(5,13)	1.51	0.46	(5,7)	1.54
pgi	0.90	0.53	(5,3)	1.70	0.53	(5,3)	1.70
cray	1.13	0.70	(15,5)	1.61	0.69	(15,15)	1.63





Conclusions and Future Work

- **Ongoing Work**
  - More end-to-end Application Studies
  - Continued Evaluation of Online Code Generation
  - Autotuning for GPUs
- **Conclusions**
  - Auto tuning can be done at many levels
    - Offline – using training runs (choices fixed for an entire run)
      - Compiler options
      - Programmer supplied per-run tunable parameters
      - Compiler Transformations
    - Online –training or production (choices change during execution)
      - Programmer supplied per-timestep parameters
      - Compiler Transformations
  - It Works!
    - Real programs run faster