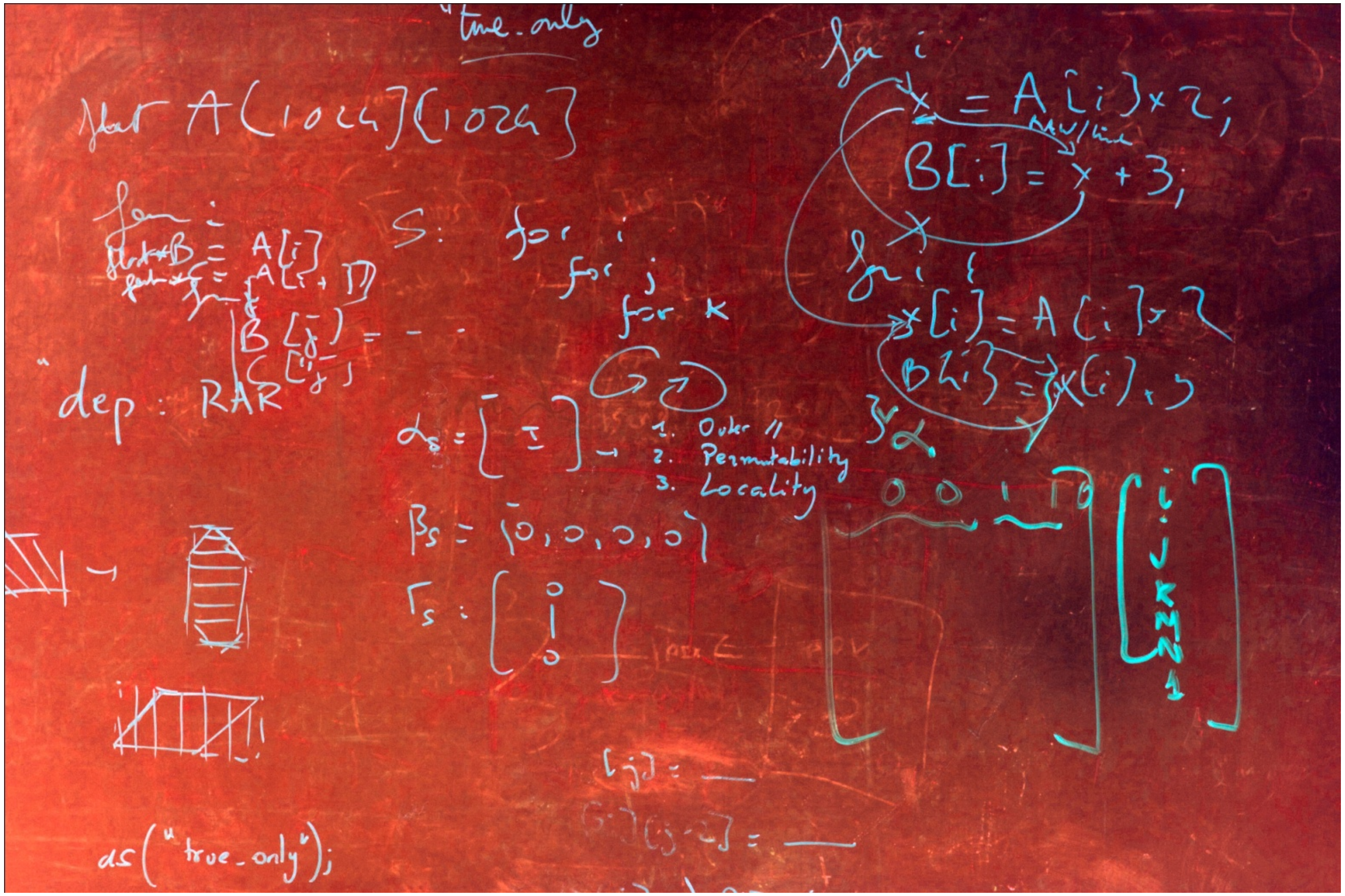
# Polyhedral Mapping Assistant and Visualizer (PUMA-V)

## Department of Energy Office of Science, Office of Nuclear Physics, SBIR/STTR Exchange Meeting, August 9-10 2016



## Polyhedral Mapping Assistant and Visualizer (PUMA-V)

### Project and Motivation

- US Department of Energy contract number DE-SC0009678:
  - Phase II STTR, 2014 Award Year.
  - April 15, 2014 – April 15, 2016 term
  - Solic. Number DE-FOA-0001019, Topic Code 39e
  - Reservoir Labs, SUNY StonyBrook, Brookhaven National Lab
- Motivation:
  - Lattice QCD community has traditionally produced very efficient home-grown software and is continuing to do so.
  - However, with the arrival of new hardware architectures, significant efforts are required to optimize the software for the target new architecture.
    - One way to deal with this is to rewrite the software to be "future-proof".
    - Another way is to see if there are any automation tools that are capable of producing efficient code for a target architecture from generic, high-level, user codes.
  - As a Phase II US DOE SBIR/STTR funded program, PUMA-V explores the second approach, seeking to use the R-Stream source-to-source compiler to optimize the Domain Wall Dirac operator implementation as well as develop accelerated solvers and state-of-the-art visualization methods
  - Columbia Physics System (CPS) as initial target.

## Polyhedral Mapping Assistant and Visualizer (PUMA-V)

### Goals for Accelerating Code Production for Nuclear Physics

- Algorithms for automatic generation of highly optimized heterogeneous code based on inherent properties of Lattice QCD problem, providing opportunities for substantial speed-ups in computation.
  - Compiler technology (R-Stream) to help with specific computations, especially those related to the proton size puzzle and muon magnetic moment, of particular interest to our domain expert partners (BNL).
- An automated visualization tool-chain to assist with software optimizations and mappings in high dimensional spaces, of great utility not only to the physics community but to the larger scientic computation enterprise.
  - Performance visualizer and IDE plugin development for intuitive code generation (Stonybrook)
- New opportunities to lower cost and power requirements for large computations needed in fundamental physics, fluid dynamics, computational biology and other scientific disciplines.
  - Faster preconditioners and solvers (CMG, Peng-Spielman, etc.)

## PUMA-V Teams

### PUMA-V Personnel

- Reservoir Labs, Compiler Technology and Solvers:
  - M. Harper Langston, PhD – PI of PUMA-V project
  - Richard Lethin, PhD - President of Reservoir Labs
  - Benoit Meister, PhD – Managing Engineer
  - Athanasios Konstantinidis, PhD – Senior Engineer
- Brookhaven National Lab, Domain Experts:
  - Taku Izubuchi, PhD
  - Meifeng Lin, PhD
  - Chulwoo Jung, PhD
- StonyBrook University, Visualization and Compiler Technology:
  - Klaus Mueller, PhD
  - Eric Papenhausen
  - Bing Wang

# Reservoir Labs, Inc.

Founded 1990 – Offices in New York City and Portland, OR

- Involved in a variety of research projects to explore ways to solve dynamic systems and effectively compile real-world algorithms. Work for and with numerous government institutions, and collaborate closely with other leading researchers and academics around the world. Sample projects:
  - R-Stream® — advanced compiler technology enables developers to create program logic once and produce optimized code for multiple parallel computing architectures.
  - R-Solve® – An automated reasoning technology that addresses dynamic problems in advanced planning and decision analysis, modeling and simulation.
  - ENSIGN – Cutting-edge hypergraph analysis technology for Big Data applications spanning security, finance and biology.
- Provide services and products to commercial clients through research technologies for organizations working on novel high-performance systems; package those technologies in turnkey commercial and government solutions that address important science and security issues. Sample products:
  - R-Scope Network Security Monitoring — Puts networks under a microscope so customers can respond to both known and zero-day attacks before becoming crises.
  - R-Check® SCA — Simplifies and accelerates SCA-compliance testing for defense communication systems worldwide, shortening the timeframe for checking from weeks and months to hours.
- More than 30 full-time researchers and engineers (half with PhDs) and a mature business development department

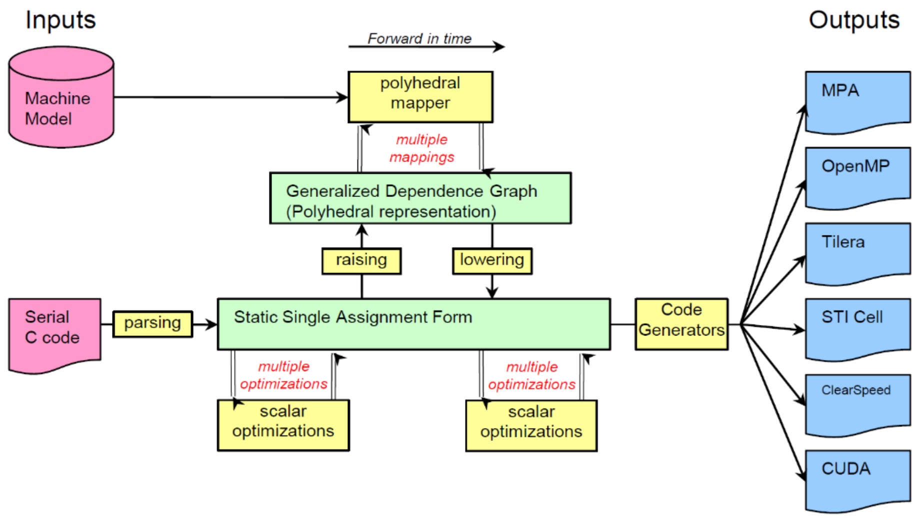
## Polyhedral Mapping Assistant and Visualizer (PUMA-V) Major Efforts

- CPS Code Modifications
- Visualizer Tool
- Extending R-Stream with an LLVM front-end for Templated C++
- Fast Linear Solvers

# R-Stream Polyhedral Model Compiler

Developed by **Reservoir Labs Inc.**

- A high-level source-to-source compiler based on the **polyhedral model**, a mathematical abstraction for analysis and transformation of computer programs:
  - Darte, Schreiber & Villard, 1985
  - Feautrier 1992
- Performs optimizations in terms of parallelization, memory manage-ment, locality etc. and can target a range of hardware architectures.
- Accepts a sequential C program as input and produces code in a variety of formats, including C + **OpenMP** and **CUDA**.
  - Meister et. al, 2011
  - Vasilache et. Al, 2013
- Used for PUMA-V in targeting key code kernels in the LQCD formulations



## CPS Code Modification Motivation

- In Lattice QCD (LQCD) simulations, the most computation intensive part is the inversion of the fermion Dirac matrix, $M$ .
  - In quark propagator calculation, need to solve $M\varphi = b$.
  - In gauge ensemble generation, need to solve $M^\dagger M\chi = \eta$.

- The recurring component of the matrix inversions is the application of the Dirac matrix on a fermion vector.

- For Wilson fermions, the Dirac matrix can be written as

$$M = 1 - \kappa D, \qquad (1)$$

up to a normalization factor, where $\kappa$ is the hopping parameter, and $D$ is the derivative part of the fermion matrix, the Dslash operator.

- The matrix-vector multiplication in LQCD essentially reduces to the application of the Dslash operator on a fermion vector.

- The motivations for this work are
  - to see if source-to-source code generators can produce reasonably performant code if only given a naive implementation of the Dslash operator as an input;
  - to investigate optimization strategies in terms of SIMD vectorization, OpenMP multithreading and multinode scaling with MPI.

## Domain Wall Dslash Operator

- The Domain Wall (DW) fermion matrix can be written as

$$M^{DW}_{x,s;x',s'} = (4 - m_5)\delta_{x,x'}\delta_{s,s'} - \frac{1}{2}D^{W}_{x,x'}\delta_{s,s'} + D^{5}_{s,s'}\delta_{x,x'}\,, \qquad (2)$$

where $m_5$ is the domain wall height, $D^W_{x,x'}$ is the Wilson Dslash operator, and $D^5_{ss'}$ is the fermion mass term that couples the two boundaries in the 5th dimension,

$$\begin{aligned} D^5_{ss'} &= -\frac{1}{2}\left[(1-\gamma_5)\delta_{s+1,s'} + (1+\gamma_5)\delta_{s-1,s'} - 2\delta_{s,s'}\right] \\ &\quad + \frac{m_f}{2}\left[(1-\gamma_5)\delta_{s,L_s-1}\delta_{0,s'} + (1+\gamma_5)\delta_{s,0}\delta_{L_s-1,s'}\right]. \end{aligned} \qquad (3)$$

- Most FLOPs are in the 4D derivative term (DWF 4D Dslash) in Eq.(2): 1320 flops per site.
- $\longleftrightarrow$focus of our optimizations.

## R-Stream Transformation of the DW 4D Dslash

- The input code we used is the unoptimized `noarch` implementation of the Dslash in CPS.
- Most straightforward implementation, direct transcription of the Dslash definition.
- Some manual code transformation was needed to get R-Stream to parse the code:
  - Delinearized array access: 1D array → multidimensional array
  - Removal of the modulo statements: introduced boundary padding.

- With these changes, R-Stream was able to produce generated code. However, the resulting code did not give very good performance. Some hand tuning was required.
- Remainder involved hand-tuning efforts.

## OpenMP Optimization: Multithreading with OpenMP

- Within the node, we use OpenMP for multithreading.
- Three strategies have been explored:
  - Simple Pragma: Thread the outer loop, usually the $t$ loop.
    $\hookrightarrow$ Parallelism is limited by the $t$ dimension size, won't scale well in many-core systems.
  - Compressed Loop: Compress the nested loops into one single loop.
  - Explicit Work Distribution: Similar to Compressed Loop, but explicitly assign work to each thread.

```
#pragma omp parallel
  {
   int nthreads = omp_get_num_threads();
   int tid = omp_get_thread_num();
   int work = NT*NZ*NY*(NX/2)/nthreads;
   int start = tid * work;
   int end = (tid+1) * work;
   for(lat_idx = start; lat_idx < end; lat_idx++)
   ......
  }
```

# OpenMP Optimization: OpenMP Performance

Performance was measured on LIRED, with dual-socket Haswell per node @ 2.6 GHz (24 cores).

- $8^4 \times 8$

| Num. Threads | Simple Pragma | Compressed Loop | Explicit Dist. |
|---|---|---|---|
| 1 | 28.4 GF/s | 28.0 GF/s | 28.0 GF/s |
| 2 | 51.5 GF/s | 54.1 GF/s | 54.1 GF/s |
| 4 | 90.1 GF/s | 90.1 GF/s | 90.1 GF/s |
| 8 | 135.2 GF/s | 135.2 GF/s | 144.2 GF/s |
| 16 | 127.2 GF/s | 180.2 GF/s | 154.4 GF/s |

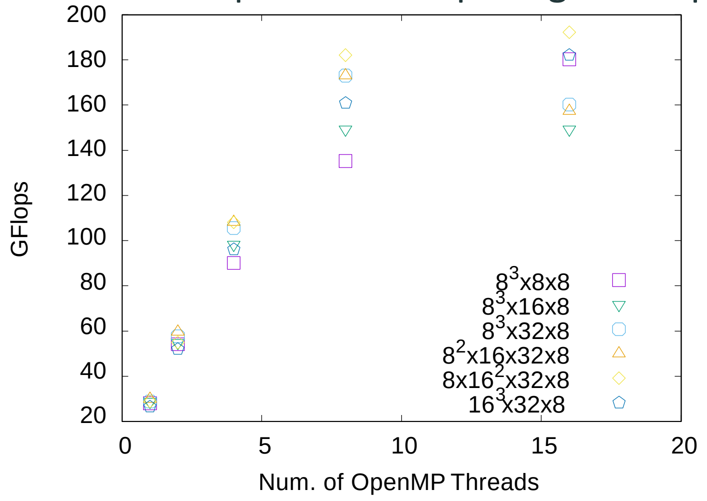
- $16^3 \times 32 \times 8$:

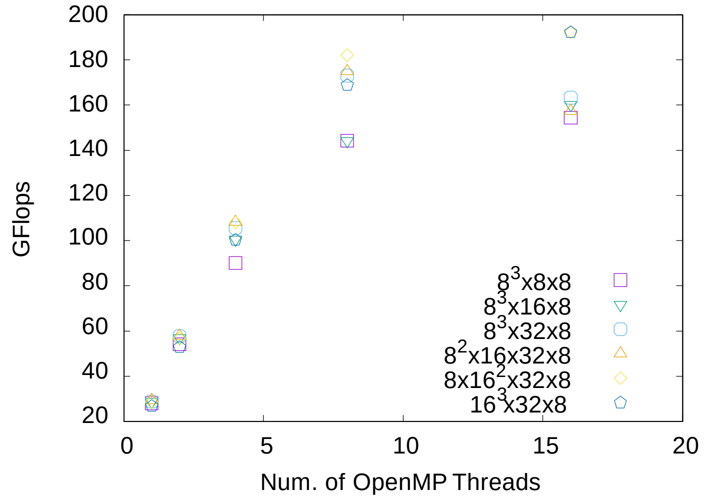
| Num. Threads | Simple Pragma | Compressed Loop | Explicit Dist. |
|---|---|---|---|
| 1 | 26.9 GF/s | 26.5 GF/s | 26.8 GF/s |
| 2 | 54.5 GF/s | 52.0 GF/s | 52.8 GF/s |
| 4 | 100.3 GF/s | 96.1 GF/s | 100.3 GF/s |
| 8 | 168.8 GF/s | 160.9 GF/s | 168.8 GF/s |
| 16 | 197.7 GF/s | 182.1 GF/s | 192.2 GF/s |

## OpenMP Optimization: OpenMP Summary

- Three threading approaches result in similar performances, except when the problem size is small, Simple Pragma doesn't scale as well.
- Surprisingly, the performance does not deteriorate with a much larger lattice size $\leftrightarrow$possible indication of poor cache reuse.
- Volume comparison:

Left - Compressed Loop. Right - Explicit Work Distribution.



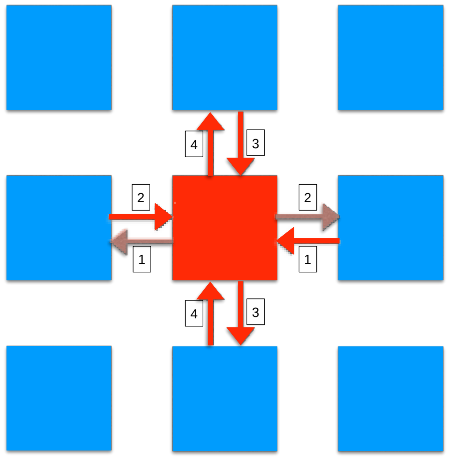


We also found that that binding OpenMP threads to the processors can improve the OpenMP performance a lot. With gcc, this is done through

```
export OMP_PROC_BIND=true
```

## Multinode/MPI Optimization: Internode Communication

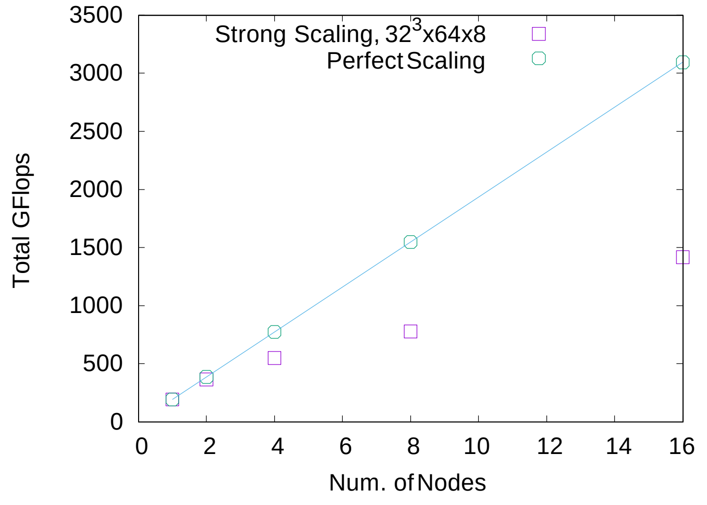
- We use QMP for communications between nodes.
- The communication pattern is illustrated in the following. There is blocking for each transfer sequence.

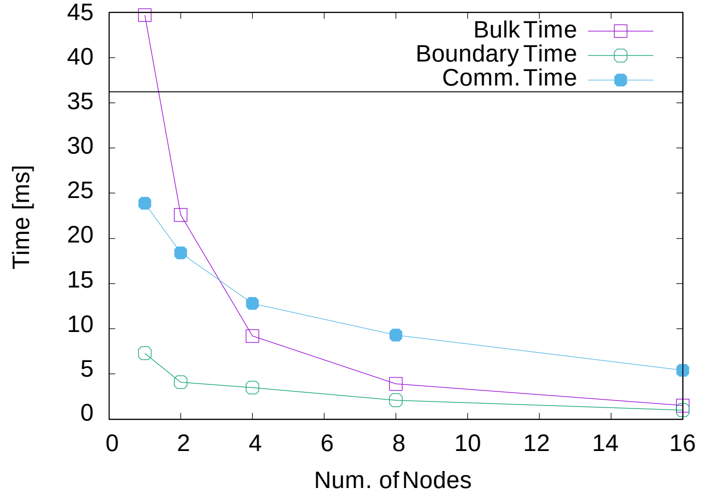


- The best performance is obtained with 2 MPI processes per node (1 MPI process per socket, improved data locality).
- With each MPI process, a number of threads equal to the number of compute cores are used.
- We dedicate one thread (the `master` thread) to do the communications, and the rest of the threads for computation.
- Do bulk computation first while waiting for the communication to complete, then do the boundary computation.

# Multinode/MPI Optimization: Multinode Performance

- Strong scaling study of a $32^3 \times 64 \times 8$ calculation was performed on LIRED, with dual-socket Intel Haswell CPUs and Mellanox 56 Gigabit FDR interconnect.
- The performance scales well up to 4 nodes, and scales sublinearly from 8 to 16 nodes.
- After 4 nodes, the total time is dominated by the communication time.
- Bulk computation itself scales well with the number of nodes.
- Rediscovered the old truth: Communication is the bottleneck for strong scaling!





## CPS Code Modification Conclusions

- To produce efficient Dslash code, optimizations in terms of data layout, SIMD, OpenMP scaling and internode communications have been studied.

- By vectorizing and changing the memory access pattern, we obtained 34% peak single-core performance in single precision.
  <⟶May still have poor cache reuse.

- On single node, OpenMP scaling deteriorates after 16 threads.
  <⟶Further improvements possible.

- Multinode strong scaling is limited by the communication cost.
  <⟶Better (higher-bandwidth) interconnects are critical.

  - Results in this presentation were obtained using the high-performance Handy and LIRED computing systems at the Institute for Advanced Computational Science at Stony Brook University and the hpc1 computing cluster at the Brookhaven National Laboratory.

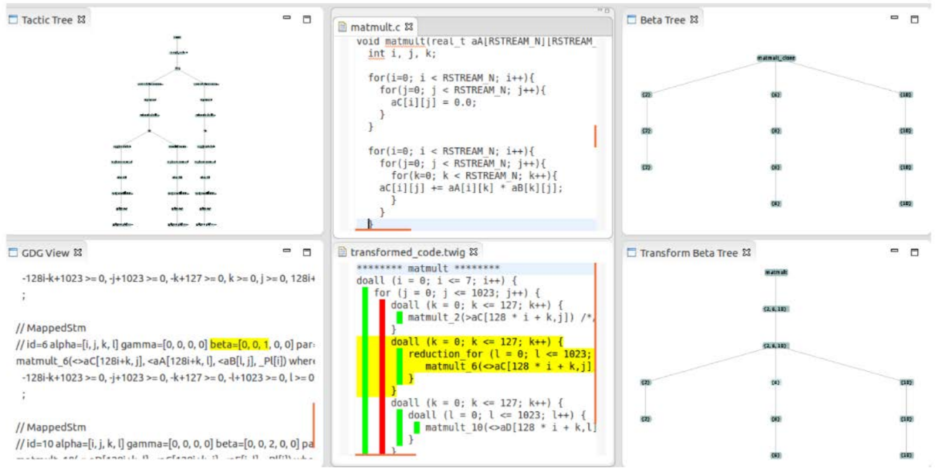
## Polyhedral Mapping Assistant and Visualizer (PUMA-V) Major Efforts

- CPS Code Modifications
- Visualizer Tool
- Extending R-Stream with an LLVM front-end for Templated C++
- Fast Linear Solvers

## PUMA-V Visualizer: Previous Version

Visualizer Eclipse Plugin with Performance Tuning (Stonybrook Focus)

- Automated toolchain to allow user-in-the-loop to make more intuitive decisions for parallel-ization and R-Stream compiler optimizations. (Video)



matmult.c

```
void matmult(real_t aA[RSTREAM_N][RSTREAM_
  int i, j, k;

  for(i=0; i < RSTREAM_N; i++){
    for(j=0; j < RSTREAM_N; j++){
      aC[i][j] = 0.0;
    }
  }

  for(i=0; i < RSTREAM_N; i++){
    for(j=0; j < RSTREAM_N; j++){
      for(k=0; k < RSTREAM_N; k++){
    aC[i][j] += aA[i][k] * aB[k][j];
      }
    }
  }
```

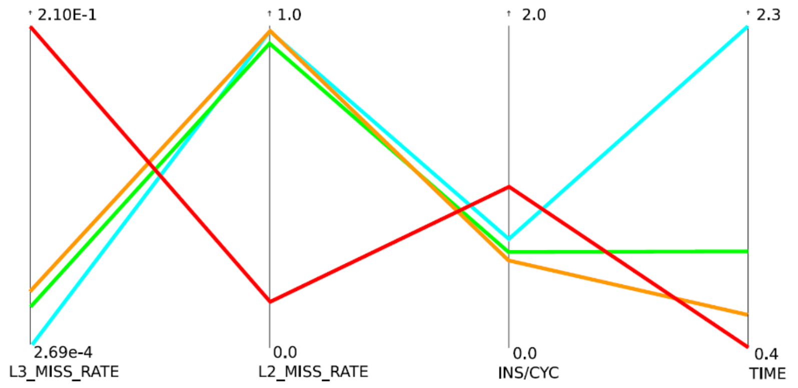
GDG View

```
  -128i-k+1023 >= 0, -j+1023 >= 0, -k+127 >= 0, k >= 0, j >= 0, 128i+
  ;

// MappedStm
// id=6 alpha=[i, j, k, l] gamma=[0, 0, 0, 0] beta=[0, 0, 1, 0, 0] par:
matmult_6(<>aC[128i+k, j], <aA[128i+k, l], <aB[l, j], _Pl[i]) where
  -128i-k+1023 >= 0, -j+1023 >= 0, -k+127 >= 0, -l+1023 >= 0, l >= 0
  ;

// MappedStm
// id=10 alpha=[i, j, k, l] gamma=[0, 0, 0, 0] beta=[0, 0, 2, 0, 0] pa
```

transformed_code.twig

```
******** matmult ********
doall (i = 0; i <= 7; i++) {
  for (j = 0; j <= 1023; j++) {
    doall (k = 0; k <= 127; k++) {
      matmult_2(>aC[128 * i + k,j]) /*
    }
    doall (k = 0; k <= 127; k++) {
      reduction_for (l = 0; l <= 1023;
        matmult_6(<>aC[128 * i + k,j]
      }
    }
    doall (k = 0; k <= 127; k++) {
      doall (l = 0; l <= 1023; l++) {
        matmult_10(<>aD[128 * i + k,l]
      }
    }
```

- Performance Profiling



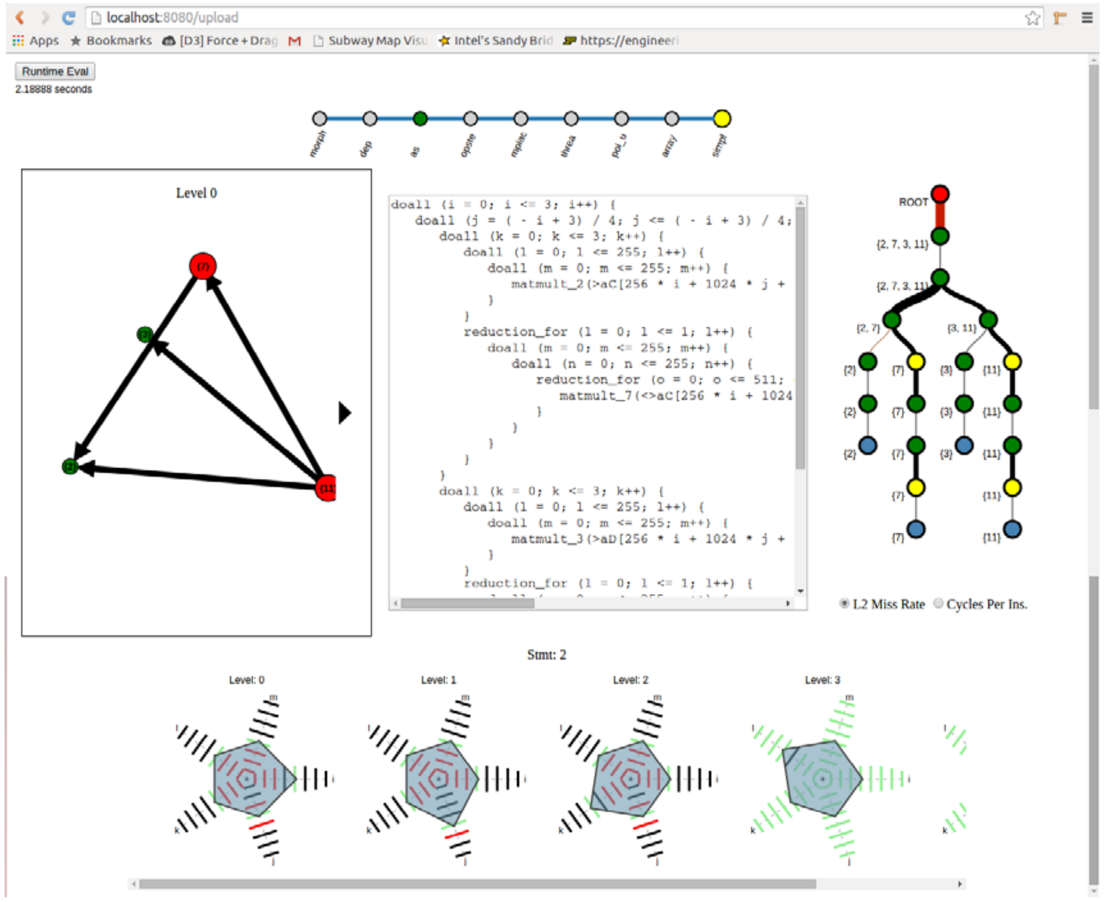
- **Tool and results published Proceedings of IEEE VISSOFT 2015**

## PUMA-V Visualizer Current Version and Updates

### Visualizer Browser Plugin with Performance Tuning

- New implementation using the d3js javascript visualization library.
- New list of tactics view is now a subway visualization.
- Beta tree nodes are colored to show the level of parallelism.
- Strongly Connected Components graph includes more visualizations, giving more insight into locality.
- New code view has text highlighting to show the differences between different transformations.
- New version augments the beta tree view to visualize run time performance data.
- User now allowed to manually permute loops by dragging and dropping nodes in the beta tree.
- New star plot view allows the user to have more control over the alpha component of the schedule matrix.
- Currently optimized and tested with Google Chrome.
- Being spearheaded by Eric Papenhausen and Klaus Mueller at Stonybrook

# PUMA-V Visualizer Matrix Mult. Example

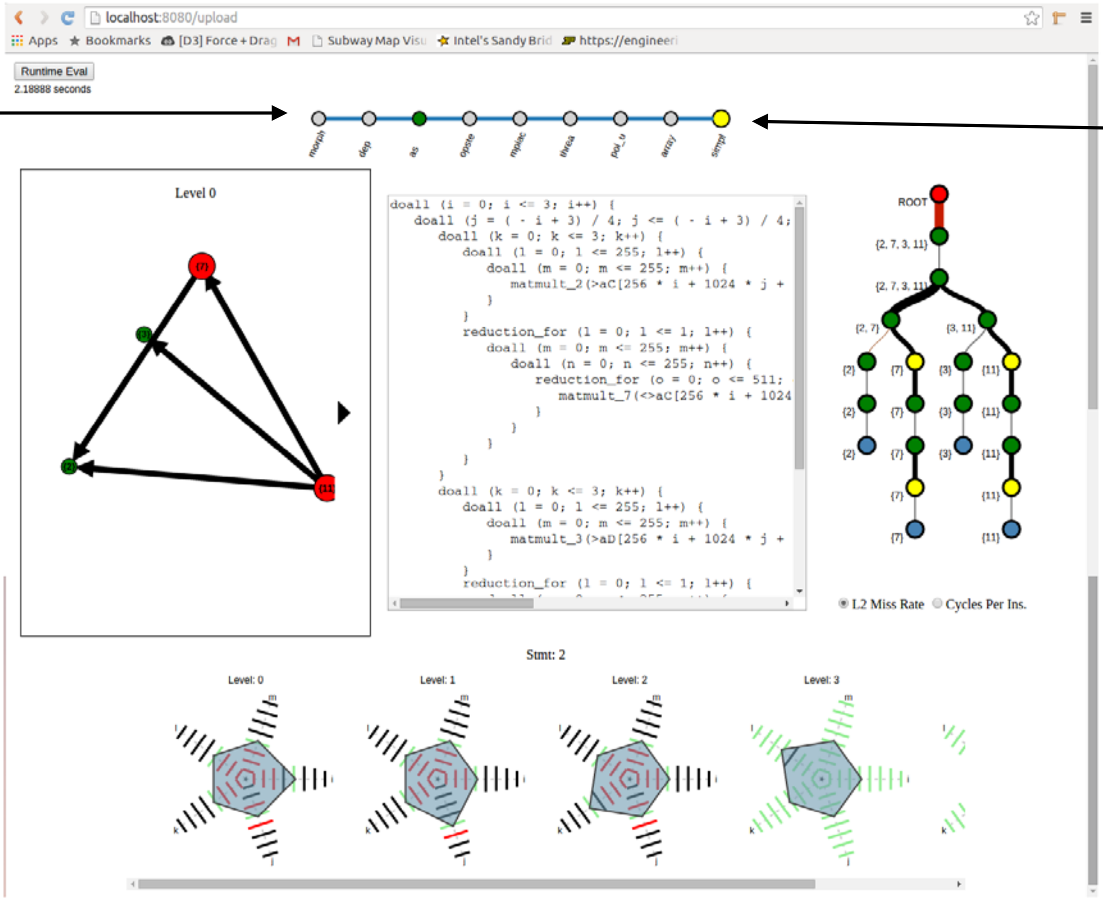


```
doall (i = 0; i <= 3; i++) {
   doall (j = ( - i + 3) / 4; j <= ( - i + 3) / 4;
      doall (k = 0; k <= 3; k++) {
         doall (l = 0; l <= 255; l++) {
            doall (m = 0; m <= 255; m++) {
               matmult_2(>aC[256 * i + 1024 * j +
            }
         }
         reduction_for (l = 0; l <= 1; l++) {
            doall (m = 0; m <= 255; m++) {
               doall (n = 0; n <= 255; n++) {
                  reduction_for (o = 0; o <= 511;
                     matmult_7(<>aC[256 * i + 1024
                  }
               }
            }
         }
      }
      doall (k = 0; k <= 3; k++) {
         doall (l = 0; l <= 255; l++) {
            doall (m = 0; m <= 255; m++) {
               matmult_3(>aD[256 * i + 1024 * j +
            }
         }
         reduction_for (l = 0; l <= 1; l++) {
```

## PUMA-V Visualizer Matrix Multiplication Example

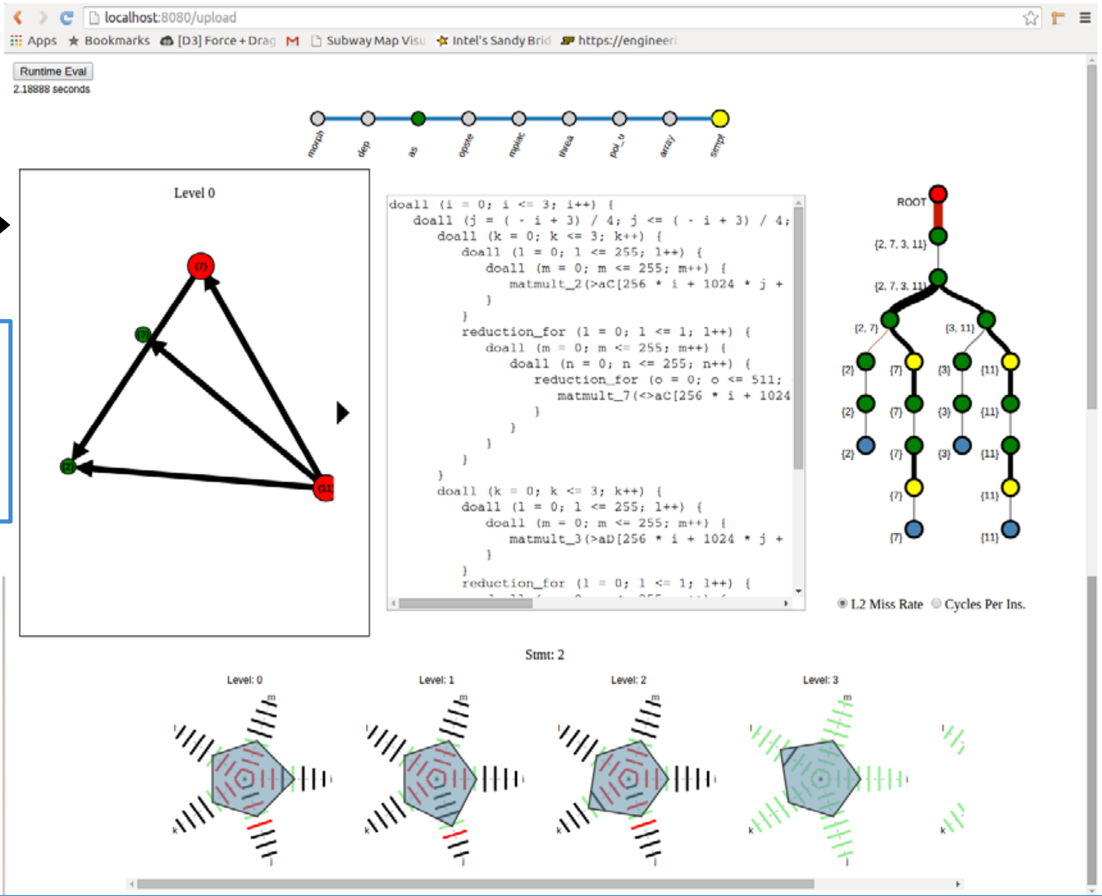
List of R-Stream tactics applied.

Yellow indicates views showing results after this tactic.



```
doall (i = 0; i <= 3; i++) {
   doall (j = ( - i + 3) / 4; j <= ( - i + 3) / 4;
      doall (k = 0; k <= 3; k++) {
         doall (l = 0; l <= 255; l++) {
            doall (m = 0; m <= 255; m++) {
               matmult_2(>aC[256 * i + 1024 * j +
            }
         }
         reduction_for (l = 0; l <= 1; l++) {
            doall (m = 0; m <= 255; m++) {
               doall (n = 0; n <= 255; n++) {
                  reduction_for (o = 0; o <= 511;
                     matmult_7(<>aC[256 * i + 1024
                  }
               }
            }
         }
      }
      doall (k = 0; k <= 3; k++) {
         doall (l = 0; l <= 255; l++) {
            doall (m = 0; m <= 255; m++) {
               matmult_3(>aD[256 * i + 1024 * j +
            }
         }
         reduction_for (l = 0; l <= 1; l++) {
```

## PUMA-V Visualizer Matrix Multiplication Example



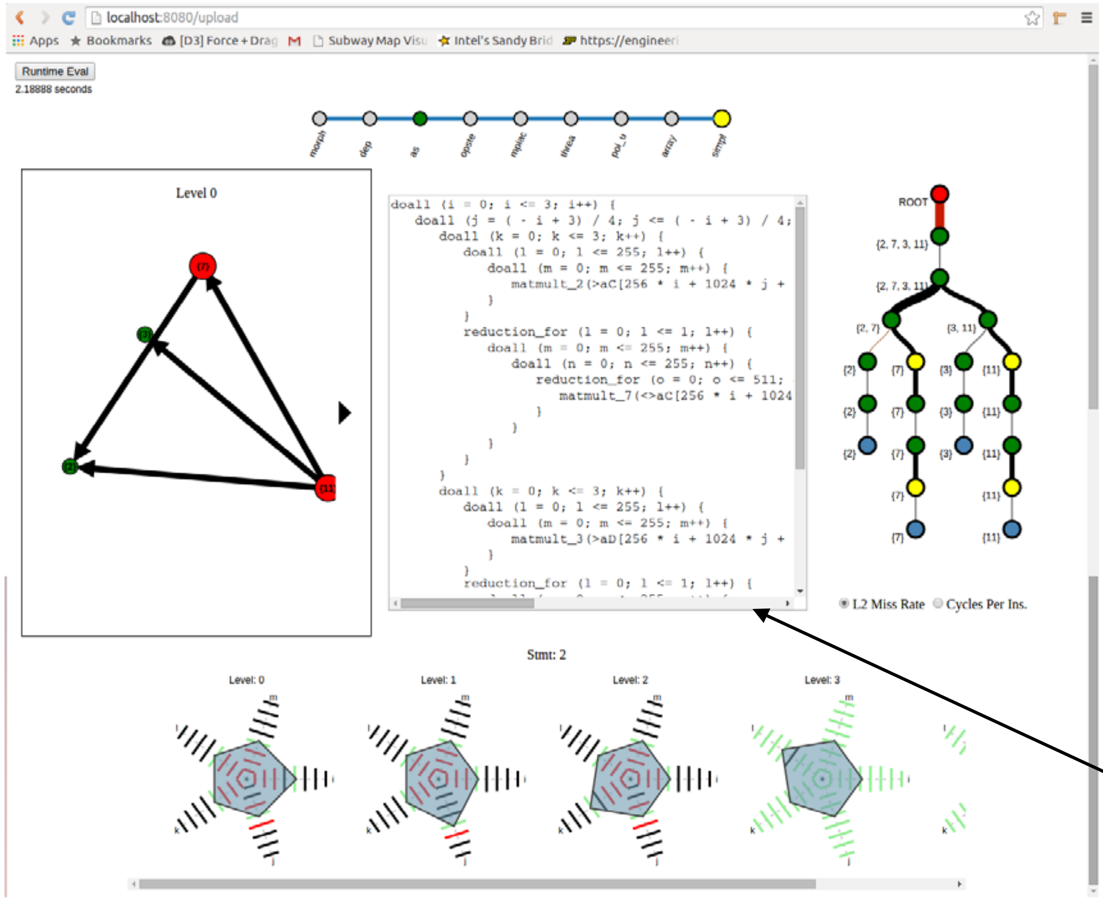
```
doall (i = 0; i <= 3; i++) {
   doall (j = ( - i + 3) / 4; j <= ( - i + 3) / 4;
      doall (k = 0; k <= 3; k++) {
         doall (l = 0; l <= 255; l++) {
            doall (m = 0; m <= 255; m++) {
               matmult_2(>aC[256 * i + 1024 * j +
            }
         }
         reduction_for (l = 0; l <= 1; l++) {
            doall (m = 0; m <= 255; m++) {
               doall (n = 0; n <= 255; n++) {
                  reduction_for (o = 0; o <= 511;
                     matmult_7(<>aC[256 * i + 1024
                  }
               }
            }
         }
      }
      doall (k = 0; k <= 3; k++) {
         doall (l = 0; l <= 255; l++) {
            doall (m = 0; m <= 255; m++) {
               matmult_3(>aD[256 * i + 1024 * j +
            }
         }
         reduction_for (l = 0; l <= 1; l++) {
```

SCC Graph constructed from Data Dependence Graph.

Nodes correspond to statements and edges indicate dependence between two statements.

- Length of edge corresponds to dependence distance.
- Width of edge corresponds to volume of data in dependence.
- Nodes colored based on level of locality with respect to inner most loop of the statement:
  - Green indicates good locality and red indicates bad locality.;
  - Size of the nodes is determined by the dependence distance of any self dependences

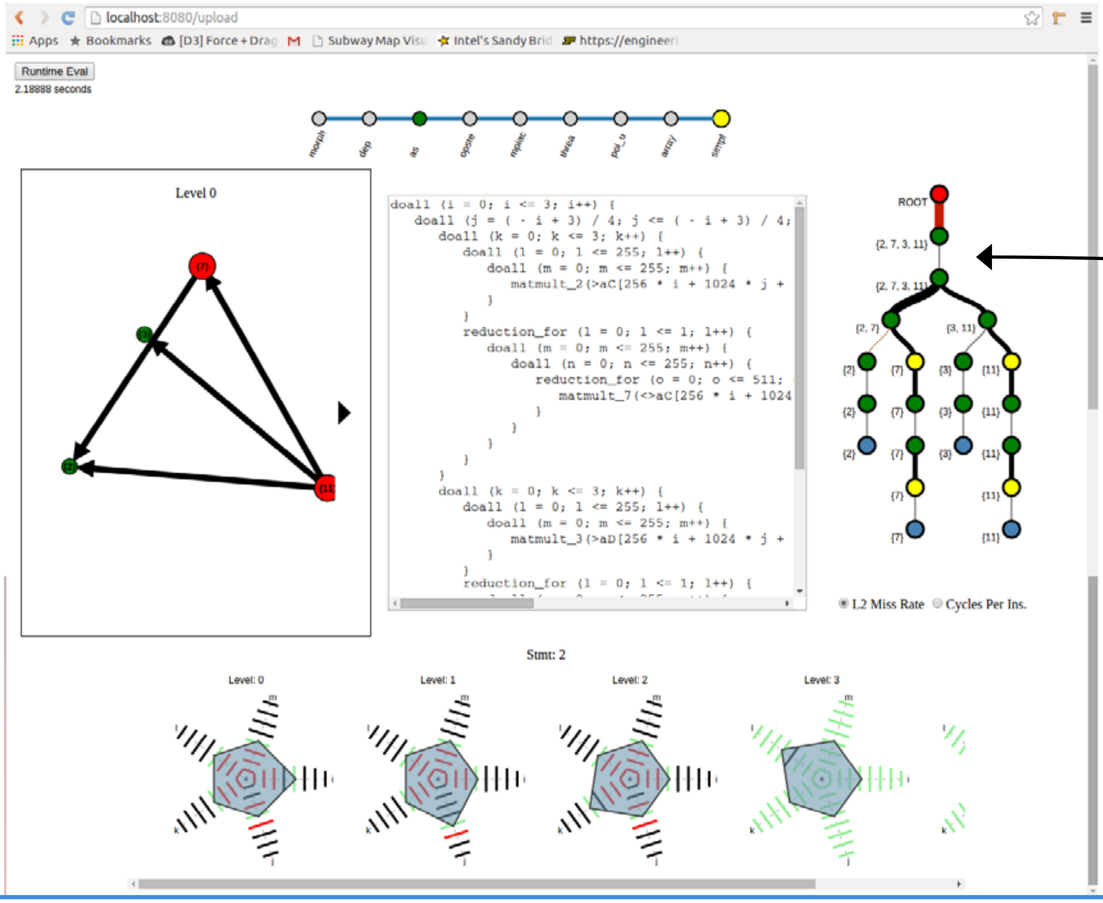
# PUMA-V Visualizer Matrix Multiplication Example



```
doall (i = 0; i <= 3; i++) {
   doall (j = ( - i + 3) / 4; j <= ( - i + 3) / 4;
      doall (k = 0; k <= 3; k++) {
         doall (l = 0; l <= 255; l++) {
            doall (m = 0; m <= 255; m++) {
               matmult_2(>aC[256 * i + 1024 * j +
            }
         }
         reduction_for (l = 0; l <= 1; l++) {
            doall (m = 0; m <= 255; m++) {
               doall (n = 0; n <= 255; n++) {
                  reduction_for (o = 0; o <= 511;
                     matmult_7(<>aC[256 * i + 1024
                  }
               }
            }
         }
      }
      doall (k = 0; k <= 3; k++) {
         doall (l = 0; l <= 255; l++) {
            doall (m = 0; m <= 255; m++) {
               matmult_3(>aD[256 * i + 1024 * j +
            }
         }
         reduction_for (l = 0; l <= 1; l++) {
```

Center view shows a pseudocode output of the code after applying the current R-Stream tactic.

# PUMA-V Visualizer Matrix Multiplication Example



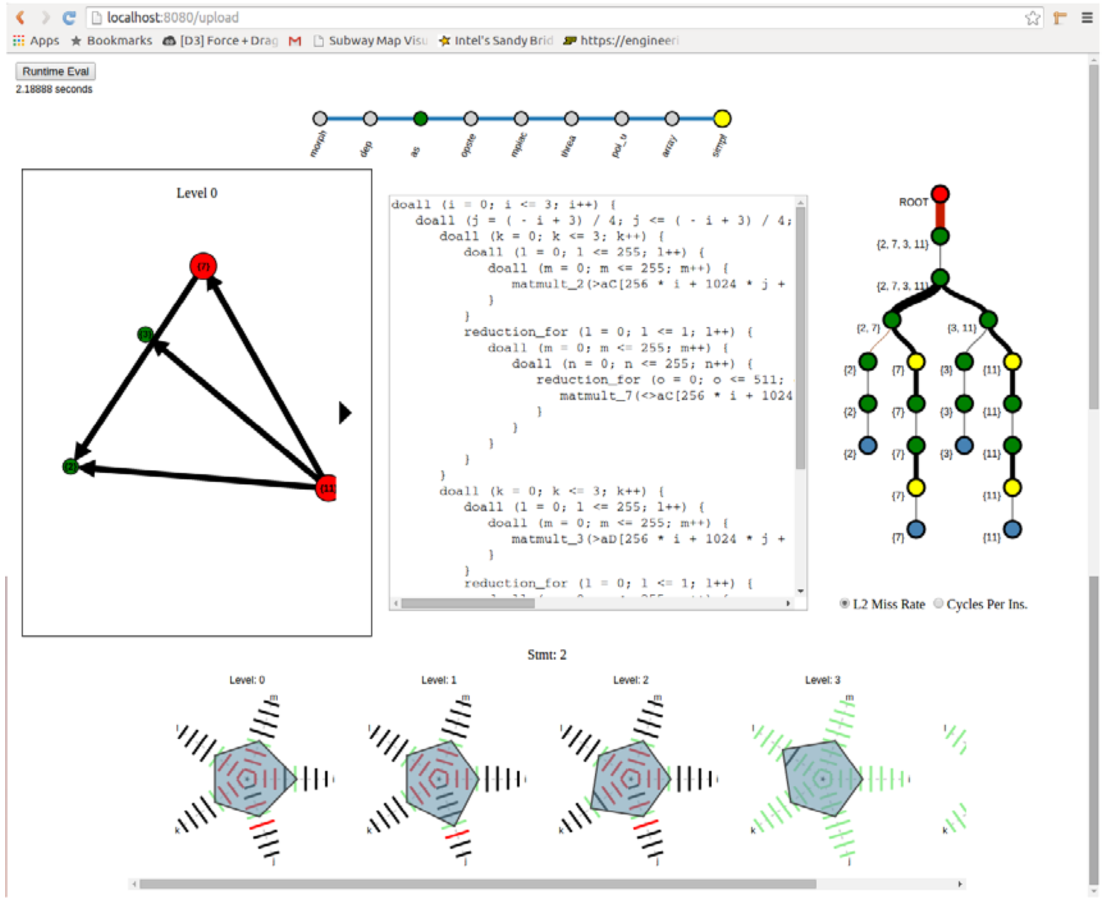
```
doall (i = 0; i <= 3; i++) {
   doall (j = ( - i + 3) / 4; j <= ( - i + 3) / 4;
      doall (k = 0; k <= 3; k++) {
         doall (l = 0; l <= 255; l++) {
            doall (m = 0; m <= 255; m++) {
               matmult_2(>aC[256 * i + 1024 * j +
            }
         }
         reduction_for (l = 0; l <= 1; l++) {
            doall (m = 0; m <= 255; m++) {
               doall (n = 0; n <= 255; n++) {
                  reduction_for (o = 0; o <= 511;
                     matmult_7(<>aC[256 * i + 1024
                  }
               }
            }
         }
      }
      doall (k = 0; k <= 3; k++) {
         doall (l = 0; l <= 255; l++) {
            doall (m = 0; m <= 255; m++) {
               matmult_3(>aD[256 * i + 1024 * j +
            }
         }
         reduction_for (l = 0; l <= 1; l++) {
```

Beta Tree View shows lexicographic ordering of loops and statements in the transformed code.

Inner nodes correspond to loops and the leaf nodes correspond to statements.

- Nodes are colored based on the level of parallelism available for the corresponding loop.
    - Nodes are either colored red, yellow, or green to indicate a sequential, reduction, or doall loop.
    - Nodes corresponding to statements are colored in blue.
- Width of edges show distribution of time spent in each branch of beta tree based on empirical performance and are colored based on the L2 cache miss rate
    - Green for low cache miss rate and red for high cache miss rate).

# PUMA-V Visualizer Matrix Multiplication Example

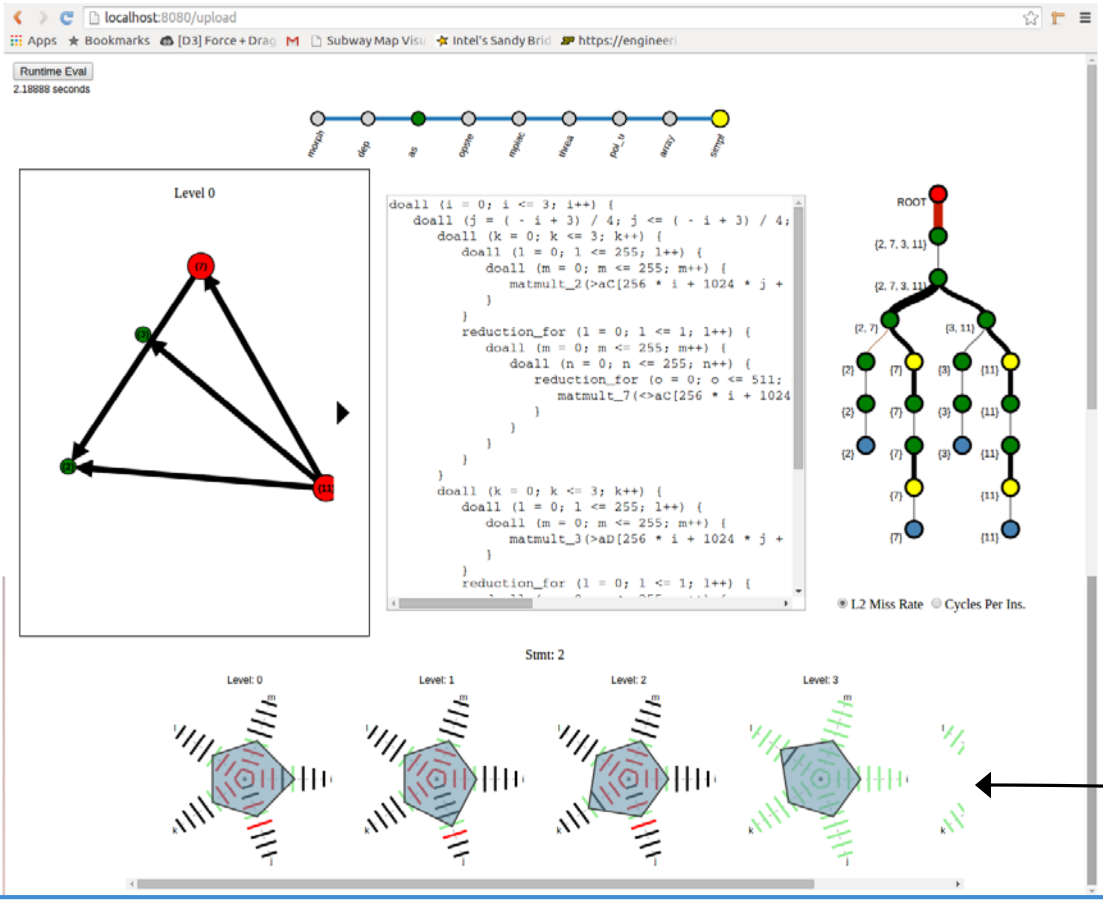


```
doall (i = 0; i <= 3; i++) {
   doall (j = ( - i + 3) / 4; j <= ( - i + 3) / 4;
      doall (k = 0; k <= 3; k++) {
         doall (l = 0; l <= 255; l++) {
            doall (m = 0; m <= 255; m++) {
               matmult_2(>aC[256 * i + 1024 * j +
            }
         }
         reduction_for (l = 0; l <= 1; l++) {
            doall (m = 0; m <= 255; m++) {
               doall (n = 0; n <= 255; n++) {
                  reduction_for (o = 0; o <= 511;
                     matmult_7(<>aC[256 * i + 1024
                  }
               }
            }
         }
      }
      doall (k = 0; k <= 3; k++) {
         doall (l = 0; l <= 255; l++) {
            doall (m = 0; m <= 255; m++) {
               matmult_3(>aD[256 * i + 1024 * j +
            }
         }
         reduction_for (l = 0; l <= 1; l++) {
```

Runtime Eval Button

When clicked, runtime evaluation will compile and execute the current transformed code, evaluate performance and update the beta tree edges with performance metrics from HPCtoolkit.

# PUMA-V Visualizer Matrix Multiplication Example



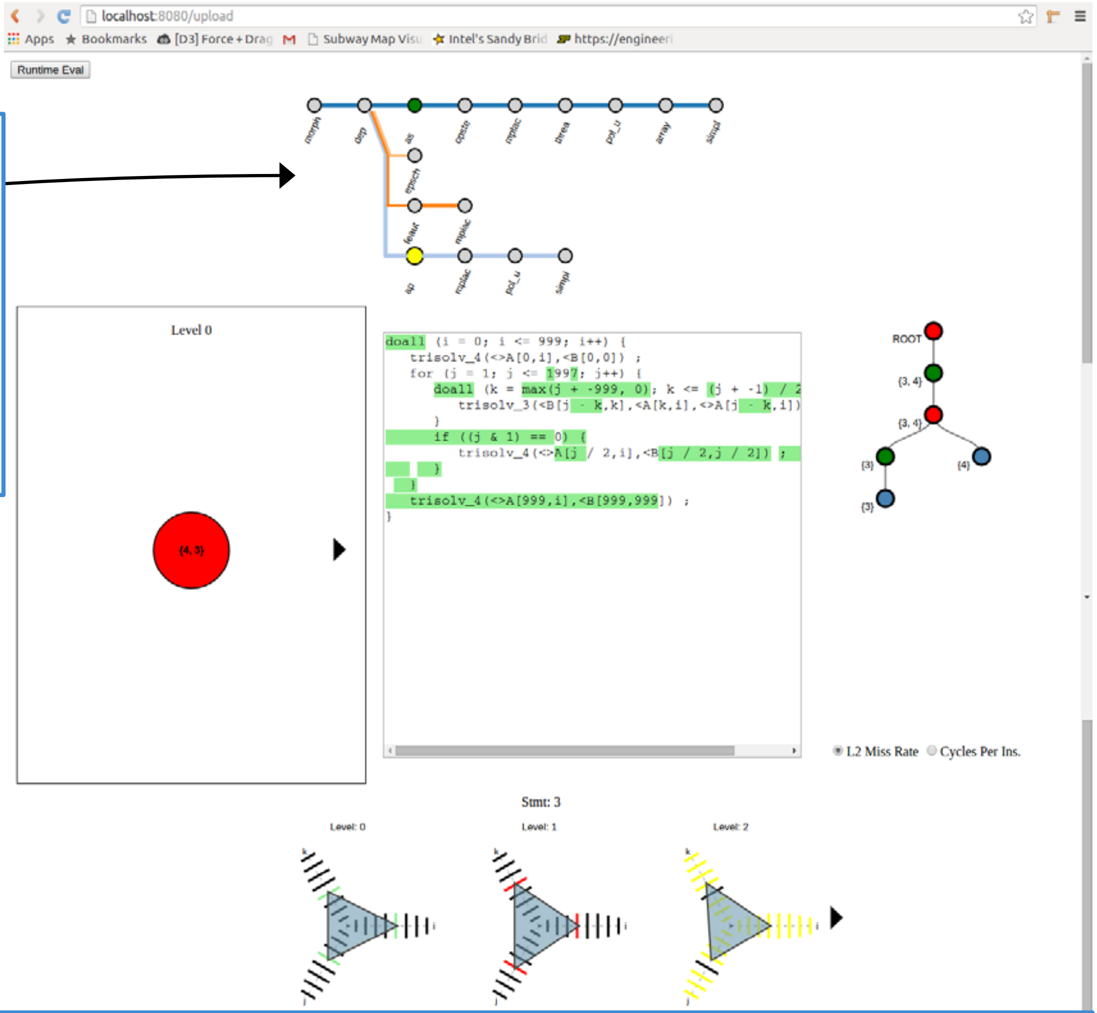
```
doall (i = 0; i <= 3; i++) {
   doall (j = ( - i + 3) / 4; j <= ( - i + 3) / 4;
      doall (k = 0; k <= 3; k++) {
         doall (l = 0; l <= 255; l++) {
            doall (m = 0; m <= 255; m++) {
               matmult_2(>aC[256 * i + 1024 * j +
            }
         }
         reduction_for (l = 0; l <= 1; l++) {
            doall (m = 0; m <= 255; m++) {
               doall (n = 0; n <= 255; n++) {
                  reduction_for (o = 0; o <= 511;
                     matmult_7(<>aC[256 * i + 1024
                  }
               }
            }
         }
      }
      doall (k = 0; k <= 3; k++) {
         doall (l = 0; l <= 255; l++) {
            doall (m = 0; m <= 255; m++) {
               matmult_3(>aD[256 * i + 1024 * j +
            }
         }
         reduction_for (l = 0; l <= 1; l++) {
```

Star plots sequence represents alpha component of schedule matrix.

Each start plot represents each loop level or loop in a nest
- Can perform transformations that have more control over the execution order such as loop interchange or skewing by dragging points on the relevant axis' in a star plot.
- Axes colored based on the parallelism of the transformation when points dragged to that axis.
  - Green indicates doall parallelism, yellow indicates a reduction and red indicates a sequential loop.
  - Black indicate that this transformation will be illegal.

## PUMA-V Visualizer Trisolv Optimization Example

Multiple different branches
in the R-Stream Tactics.
Each branch is a different set
of optimizations that are applied



```
doall (i = 0; i <= 999; i++) {
   trisolv_4(<>A[0,i],<B[0,0]) ;
   for (j = 1; j <= 1997; j++) {
      doall (k = max(j + -999, 0); k <= (j + -1) / 2
         trisolv_3(<B[j - k,k],<A[k,i],<>A[j - k,i])
      }
      if ((j & 1) == 0) {
         trisolv_4(<>A[j / 2,i],<B[j / 2,j / 2]) ;
      }
   }
   trisolv_4(<>A[999,i],<B[999,999]) ;
}
```

In this example, each branch corresponds to a different scheduling algorithm.

- Parts of the code view are also highlighted in green, showing the parts of the code that have changed since the previous tactic.

## Polyhedral Mapping Assistant and Visualizer (PUMA-V) Tool Video

Published at Sixth International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2016) and 2016 New York Scientific Data Summit (NYSDS 2016)

- Link for Sample Video:
  - https://drive.google.com/open?id=0B1lSipOr3uq8XzFDTjR1NnhsYUk

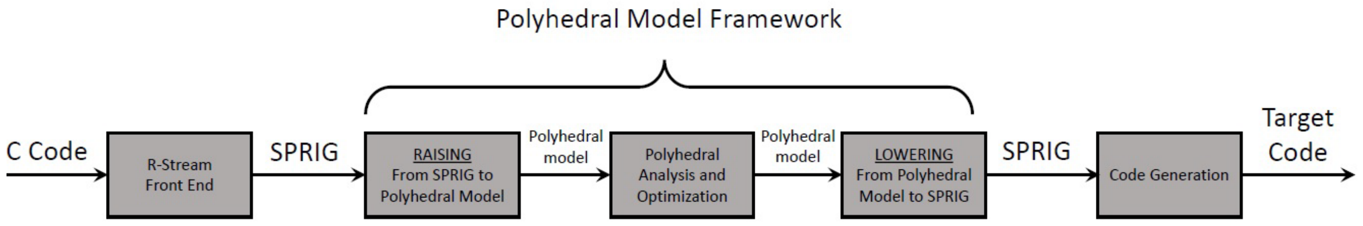
## Polyhedral Mapping Assistant and Visualizer (PUMA-V) Major Efforts

- CPS Code Modifications
- Visualizer Tool
- Extending R–Stream with an LLVM front–end for Templated C++
- Fast Linear Solvers

## Extending R-Stream with an LLVM Front-End for Templated C++

**Goal:** extending the R-Stream compiler to apply the polyhedral analysis and optimization capabilities described on templated C++ codes like qdp++, among other languages.

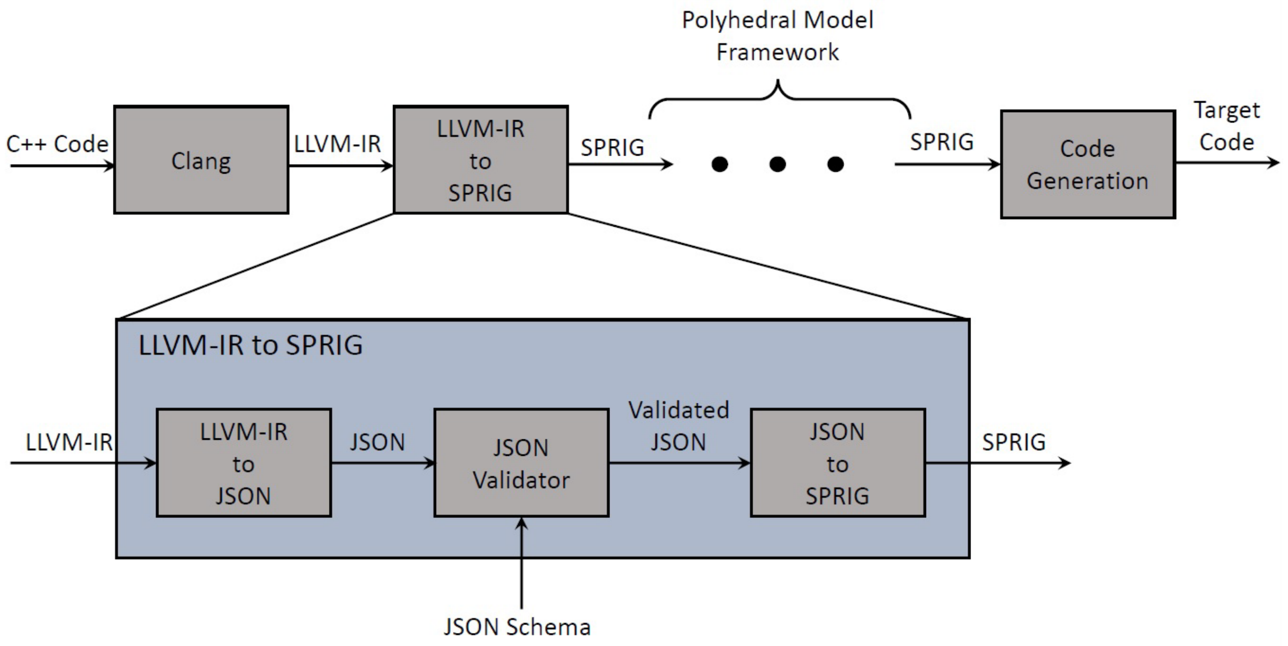
- Implemented converter from LLVM compiler Internal Representation (IR) to R-Stream's IR
- Expected result is for R-Stream to benefit from all the front-ends associated with LLVM, and in particular its Clang state-of-the-art C++ front-end
- Current R-Stream Compilation Flow


Polyhedral Model Framework
C Code
R-Stream Front End
SPRIG
RAISING From SPRIG to Polyhedral Model
Polyhedral model
Polyhedral Analysis and Optimization
Polyhedral model
LOWERING From Polyhedral Model to SPRIG
SPRIG
Code Generation
Target Code

- LLVM IR and SPRIG have a lot of commonalities. We determined that an LLVM IR to SPRIG IR translation would be the most simple and beneficial implementation path.
  - Starting from LLVM ensures support for any language accepted by an LLVM front-end.
  - Supported languages include C, C++, and Julia, and we expect that FORTRAN will also be supported within the next few years.
  - This will enable us to incorporate both LLVM and SPRIG optimizations in the compilation process. Besides, we find development within SPRIG easier than in LLVM, perhaps because its architecture is geared towards clarity rather than last-drop optimal compilation time.

## Extending R-Stream with an LLVM Front-End for Templated C++

New R-Stream flow includes an LLVM IR to SPRIG conversion step.



- The detail of this step is represented at the bottom of the figure.
- Basically serialize LLVM IR objects using the JSON textual object representation, and re-serialize them as Java objects within R-Stream.
- JSON is a language-independent human-readable open-standard data format
  - Attractive for achieving our goal because it is lightweight, language-independent and numerous tools to parse, analyze and generate JSON are available in a plethora of programming languages.

## Extending R-Stream with an LLVM Front-End for Templated C++

Working towards extending the R-Stream compiler such that it will be able to apply the polyhedral analysis and optimization capabilities templated C++ codes such as QDP++.

- We have been able to successfully generate and validate JSON output from Clang for the QDP++ Dslash test function, which while quite small, embeds a complex and deep template structure. Basically serialize LLVM IR objects using the JSON textual object representation, and re-serialize them as Java objects within R-Stream.
- We continue to work to developing the full pipeline for source-to-source compilation of these complex codes through R-Stream .
- We will soon be able to translate enough LLVM IR to fully support C99. A small set of features is missing to fully support C++
  - Because R-Stream is a source-to-source compiler, supporting a new feature means extending the internal representation and implementing pretty-printing (writing back C or C++ from SPRIG) for it.

## Polyhedral Mapping Assistant and Visualizer (PUMA-V) Major Efforts

- CPS Code Modifications
- Visualizer Tool
- Extending R-Stream with an LLVM front-end for Templated C++
- Fast Linear Solvers

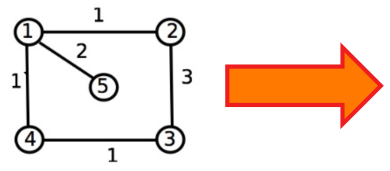
# Spectral Support Preconditioning and Nearly-Linear Time Solvers, Combinatorial Multigrid

Solving the linear system Ax = b with actual solution x := $A^{-1}$ b.

- Find B that approximates A in a spectral sense, so solving By = c is easier
- Based on Spielman and Teng (2003-current), solve a system of linear equations with a symmetric diagonally dominant (SDD) discrete operator:

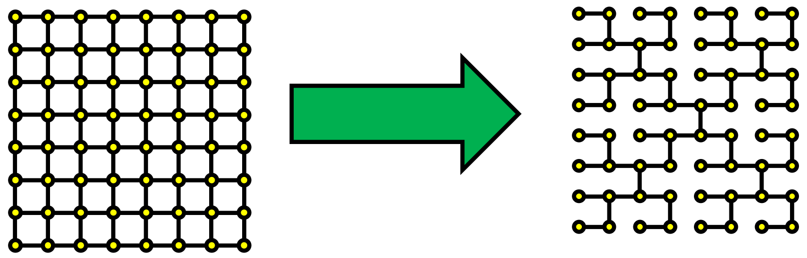
$$A\mathbf{x} = \mathbf{b}, \quad A \in \mathcal{R}^{n\times n} \quad A = A^T, \quad A_{ii} \geq \sum_{i\neq j} |A_{ij}|$$

- SDD systems have clear connection with graphs and Laplacians:



$$L_G = D - W = \begin{bmatrix} 4 & -1 & 0 & -1 & -2 \\ -1 & 4 & -3 & 0 & 0 \\ 0 & -3 & 4 & -1 & 0 \\ -1 & 0 & -1 & 2 & 0 \\ -2 & 0 & 0 & 0 & 2 \end{bmatrix}$$

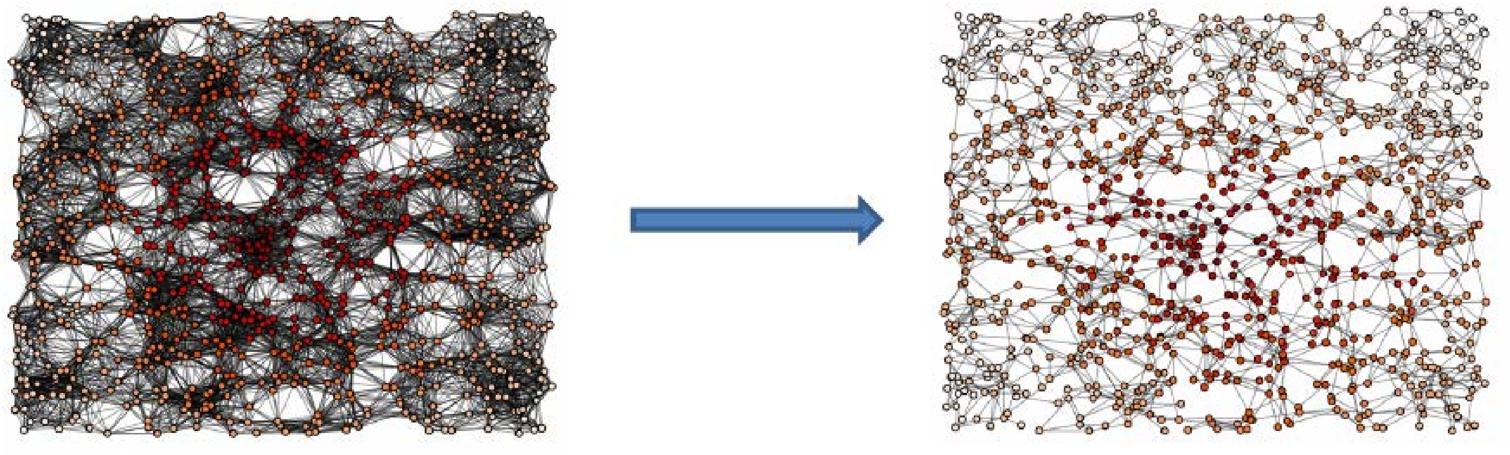
- Low-stretch trees approximate most distances to within O(log m) using only m-1 edges:



- Approaches incorporate elements of direct and iterative solvers for class of problems with graph clustering, spectral sparsification, partial factorizations, minimal trees for state of the art towards **O(mlog$^c$n)** solver for m edges and n vertices (Koutis et. al 2009: CMG, Kelner et. al 2013, Peng-Spielman 2014)

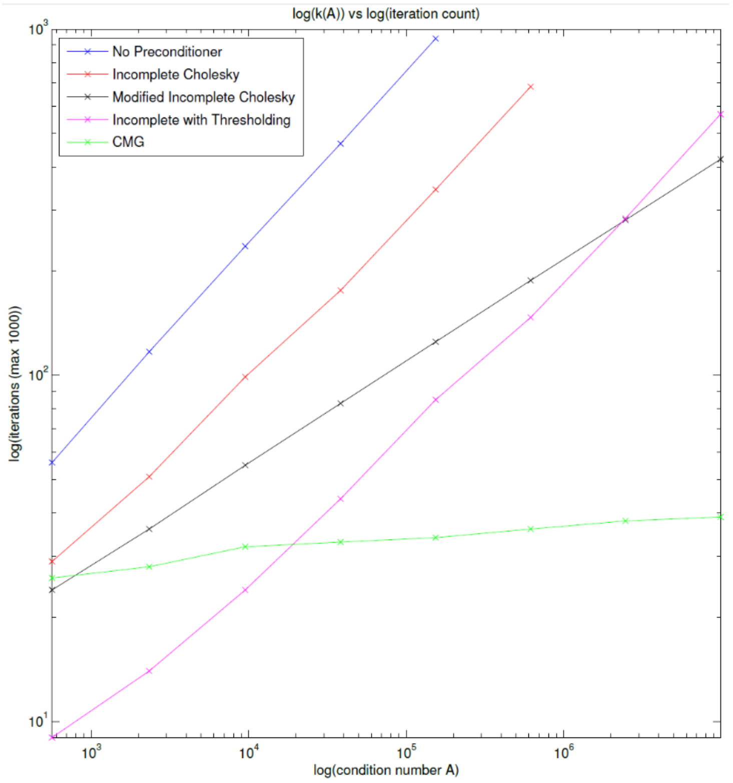
## Combinatorial Multigrid Issues

- Sparsifiers generate a "smaller" matrix or analogous graph structure, which preserves many graph parameters.



- Computationally very expensive and only seemingly practical in the theoretical realm! Further, computing on the fly tricky. The idea is to maintain "cliques" and "cycles".
  - Can be satisfied if the general structure remains in a simpler fashion using simpler strategies.
- Complex-valued data for NP problems
- NP problems not diagonally-dominant, though CMG can still handle this

## Preconditioner Sample Results



- Small non-LQCD SDD system for testing MATLAB CMG:
  - Significantly-reduced iteration counts as condition numbers grow

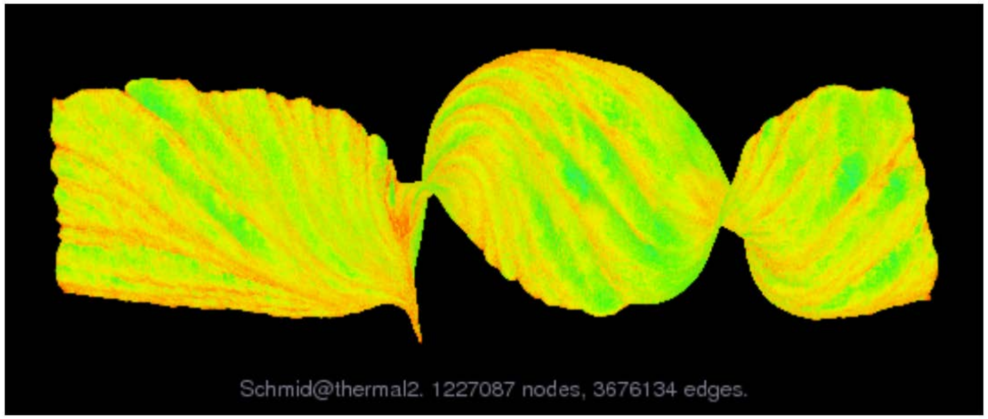
- Incomplete Cholesky-based preconditioners on small LQCD system:

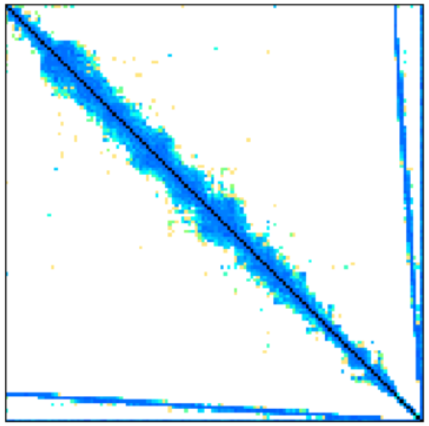
| Test Type | $N_{iter}$ | $T(s)$ |
|---|---|---|
| No Preconditioner ($P = I$) | 3352 | 35 |
| IC(0) (no-fill Incomplete Cholesky) | 1780 | 36 |
| ICT (thresholding with small values) | 22 | 6 |
| IC with diagonal compensation ($\alpha \approx 1e-4$) | 652 | 17 |

  - IC-based preconditioners show great promise, but LQCD matrices are computed at run-time, so computing incomplete factorization may require too much memory in practice

## Preconditioner Sample Results

- Unstructured FEM Problem with Sparsity:





- Results:

| $n = 1228045$, $m = 8580313$, $\kappa(A) = 7.5e6$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $S_{typ}$ | $N_{its}$ | r | $flps_{set}$ | $flps_{slv}$ | $flps_{slv}/it$ | $T_{set}$ | $T_{slv}$ | $T_{slv}/it$ |
| ICC | 10000 | $2.2e7$ | $1.2e12$ | $2.6e3$ | $1.2e8$ | $1.8e0$ | $2.6e3$ | $2.6e-1$ |
| MG | 10000 | $1.3e0$ | $2.1e7$ | $1.6e12$ | $1.6e8$ | $1.1e0$ | $4.9e3$ | $4.9e-1$ |
| CMG | 33 | $8.7e-5$ | $9.4e7$ | $9.3e9$ | $2.8e8$ | $6.7e0$ | $4.1e1$ | $1.2e0$ |

## PUMA-V Conclusions Part I

### Summary and Ongoing Goals

- **CPS Code Enhancements:** The speedup in the DWF Dslash that has been obtained as a result of this project will enable the BNL LQCD scientists to make more efficient use of the PC clusters installed at Fermilab and elsewhere for LQCD computations as well as increase the precision of their calculations, make better predictions using the QCD theory to confront experiment and potentially discover new physics in a shorter time. We are currently in the process of submitting publications based on our work and findings. We continue to work on porting our modifications and advancements to other architectures, such as Intel Xeon Phi (Knights Landing or later). At least two US supercomputers coming later this year will have KNL (Cori at NERSC and Theta at Argonne National Lab). ANL will have AURORA in 2-3 years which will have the next-generation Intel Xeon Phi architecture, so targeting these architectures will be important.
- **PUMA-V Visualizer Tool:** The two PUMA-V tools, developed for the Eclipse IDE and web-based d3js, combine fully automatic and manual techniques for optimizing source code. The techniques, scenarios, and user studies presented suggest that combining automatic methods with user intuition can lead to significantly better performance compared to automatic methods alone. Using effective visualizations through multiple views helps users seize optimization opportunities missed by the auto-parallelizing compiler. Embedding heuristics and runtime performance into the visualization assists users in identifying bottlenecks. The educational opportunities and ability to bring the polyhedral model and R-Stream to a wider audience cannot be overstated. This new tool has shown through user studies to provide better intuition to expert users and an easier entry point for novice users to the world of source-to-source optimizers and compilers.

## PUMA-V Conclusions Part II

### Summary and Ongoing Goals

- **Extending R-Stream with LLVM for Mapping C++ and QDP++ Codes:** Have worked towards extending the R-Stream compiler such that it will be able to apply the polyhedral analysis and optimization capabilities. In order to achieve this, we implemented a converter from the LLVM compiler Internal Representation (IR) to R-Stream's IR. The expected result is for R-Stream to benefit from all the front-ends associated with LLVM, and in particular its Clang state-of-the-art C++ front-end. We have been able to successfully generate and validate JSON output from Clang for the QDP++ Dslash test function, which while quite small, embeds a complex and deep template structure. We continue to work to developing the full pipeline for source-to-source compilation of these complex codes through R-Stream . We will soon be be able to translate enough LLVM IR to fully support C99. A small set of features is missing to fully support C++. Because R-Stream is a source-to-source compiler, supporting a new feature means extending the internal representation and implementing pretty-printing (writing back C or C++ from SPRIG) for it.
- **Fast Linear Solvers and Combinatorial Multigrid**: Compared multiple preconditioners such as the Incomplete Cholesky variants against the common even-odd preconditioning method for solving large LQCD systems with Conjugate Gradient. It was clear that there are strong advantages to developing new and more robust preconditioners. By reducing the iteration count and in-iteration complexities, solving the inherently large and poorly-conditioned systems with DWF could results in significant time savings.  One difficulty in our research, however, was that many of these systems are too large to fully compute and store, so algebraic solutions such as ICC are often difficult. We looked at possibilities in the realm of support graph theory. In particular, we have successfully developed a faster version of the Combinatorial Multigrid algorithm, embedding it in Petsc and test it with great success on real-world linear systems from FEM research.

## Thank You!
## Questions/Comments