From Compilers and Tools to Benchmarks and Metrics

- Seeking the Driving Forces of HPC

Chunhua “Leo” Liao

May 13, 2019

LLNL-PRES-773949

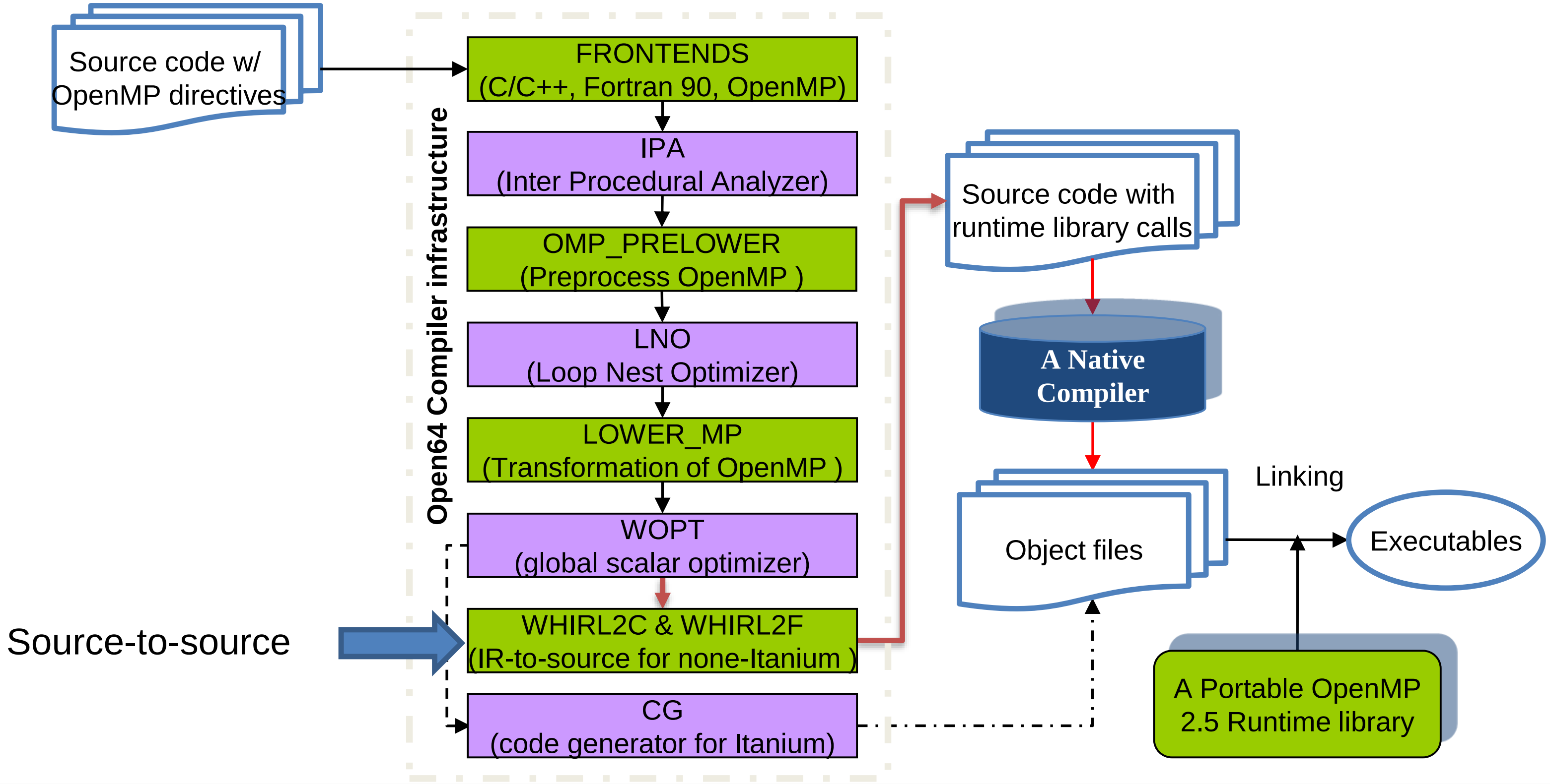
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Agenda

- OpenMP Compilers
- Tools
- Benchmarks
- Metrics
- Ongoing and Future work
- Conclusion

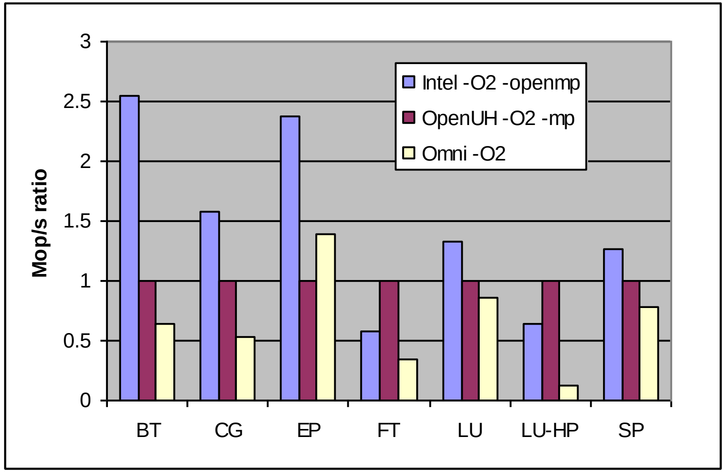
Open64+Source-to-Source => Portable OpenUH

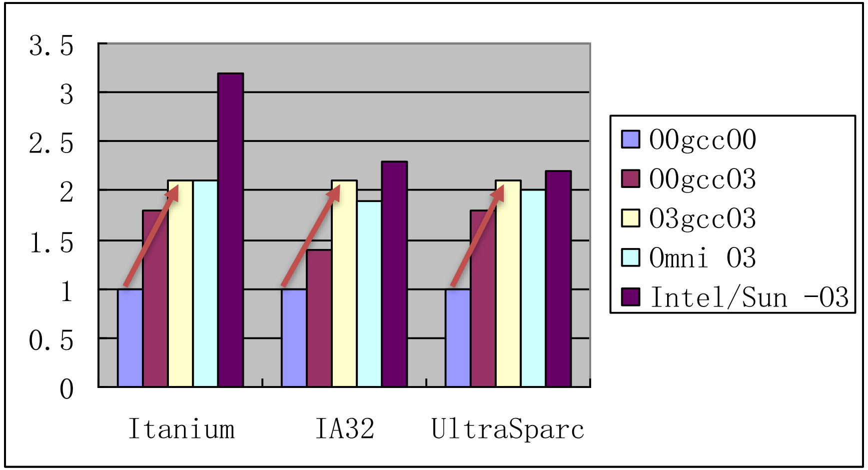


Outlining vs Inlining Translation

OpenMP Code	Classic Outlining Translation	Open64 Inlining (Nested) Translation
int main(void) { int **a,b**,c; #pragma omp parallel \\ private(c) do_sth(**a,b**,c); return 0; }	**/*Outlined function with an extra argument** **for passing addresses*/** static void __ompc_func_0(void **__ompc_args){ **int *_pp_b, *_pp_a, _p_c;** **/*dereference addresses to get shared variables */** **_pp_b=(int *)(*__ompc_args);** **_pp_a=(int *)(*(__ompc_args+1));** **/*substitute accesses for all variables*/** **do_sth(*_pp_a,*_pp_b,_p_c);** **}** int main(void){ int a,b,c; **void *__ompc_argv[2];** **/*wrap addresses of shared variables*/** ***(__ompc_argv)=(void *)(&b);** ***(__ompc_argv+1)=(void *)(&a);…** **/*OpenMP runtime call has to pass the addresses of shared variables*/** _ompc_do_parallel(__ompc_func_0, __ompc_argv); **…}**	_INT32 main() { int **a,b**,c; /*inlined (nested) microtask */ void __ompregion_main1() { _INT32 __mplocal_c; /*shared variables are keep intact, only substitute the access to private variable*/ do_sth(**a, b**, __mplocal_c); } … /*OpenMP runtime call */ __ompc_fork(&__ompregion_main1); … }

OpenMP 2.5 of OpenUH Performance: Native and Source-to-source





- Platform: Itanium 2 + RH EL AS release 3, 4 processors
- Compiler: OpenUH, Intel 8.1, Omni 1.6 using –O2
- Benchmarks: NPB 3.2 OpenMP version data set Class A

- Platforms: Itanium 2 + RH EL AS release 3, IA32 + RH9 and UltraSparc III + Solaris 9, 4-thread
- Compilers: OpenUH (source-to-source translation) and GCC [working well with optimizations due to inlining]
- Benchmark: CG of NPB 2.3 OpenMP/C

Lawrence Livermore National Laboratory

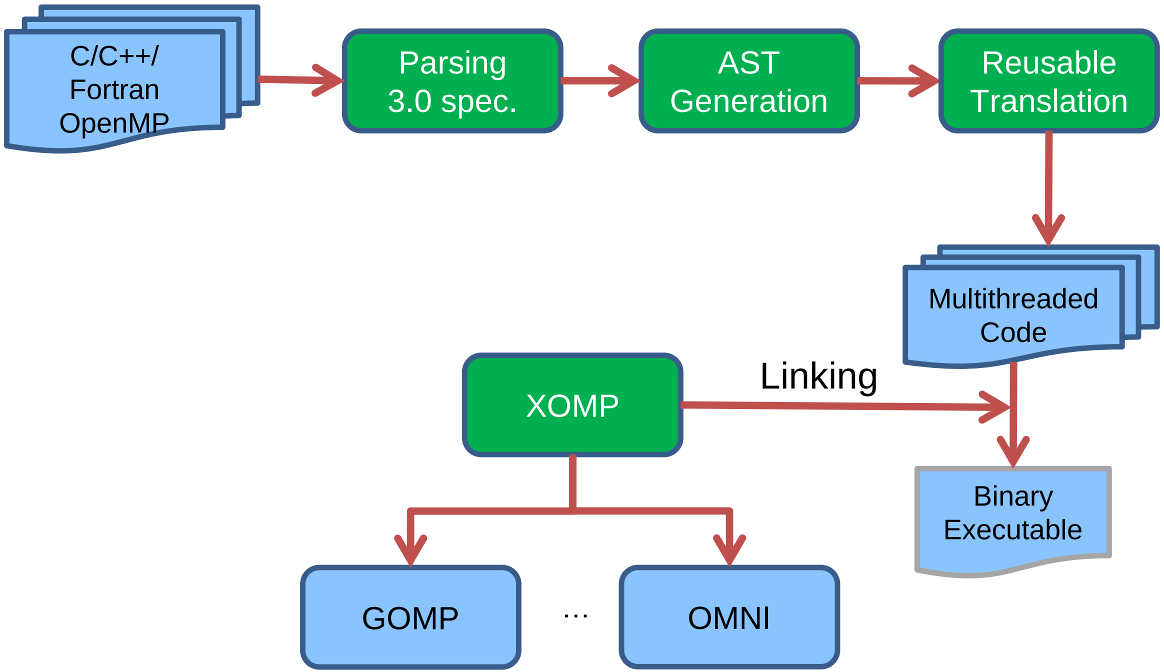
A ROSE-based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries

Chunhua Liao, Daniel J. Quinlan, Thomas Panas and Bronis R. de Supinski

Center for Applied Scientific Computing

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344

Approach: a Middle Runtime Layer (XOMP) on Top of Multiple OpenMP Runtime Libraries



How Different are OpenMP Runtime Libraries?

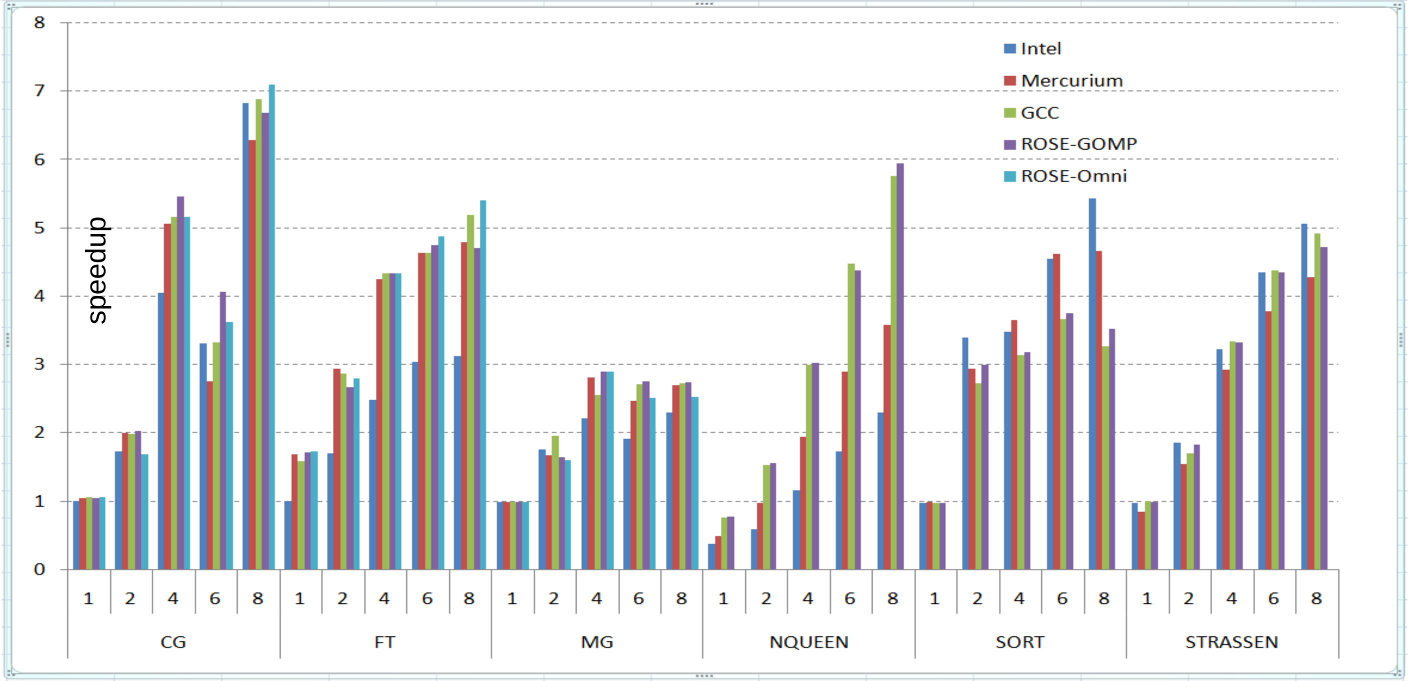
Runtime Support	GOMP	Omni
omp barrier	void GOMP_barrier (void)	void _ompc_barrier(void);
omp critical	GOMP_critical_name_start(void **data) GOMP_critical_name_end(void **data)	_ompc_enter_critical(void **data); _ompc_exit_critical(void **data);
omp single	int GOMP_single_start();	int _ompc_do_single();
omp parallel	void GOMP_parallel_start (void (*func) (void *), void *data, unsigned num_threads); void GOMP_parallel_end (void);	void _ompc_do_parallel(void (*func)(void **),void *args);
Initialization & Termination	None (Implicit)	_ompc_init(); _ompc_termination();
default loop scheduling	None (compiler generates all necessary code)	void _ompc_default_sched(int *lb, int *ub, int *step);
threadprivate	None (compiler inserts __thread)	void * _ompc_get_thdprv(void ***thdprv_p,int size,void *datap);

XOMP: a Common Translation-Runtime Layer

Rule ID	libA vs libB	XOMP interface	Compiler translation
Rule 1	funcA() and funcB(): similar functionality , but may differ by names / parameters	A common function with a union set of parameters for each	Targets XOMP_funcX()
Rule 2.1	libA has an extra funcA() : due to special need	XOMP_funcA() { if (libA) funcA(); else NOP; }	Targets XOMP_funcA()
Rule 2.2	funcA()’s functionality: suitable for runtime support e.g. default static-even loop scheduling	XOMP_funcA() { copy of funcA() body here }	Targets XOMP_funcA()
Rule 2.3	funcA() ‘s functionality: suitable for compiler translation	No XOMP function	self-contained w/o runtime support
Rule 3	Support for feature X is too different to be merged	XOMP_funcA() XOMP_funcB()	Custom translation for each

~80% compiler translations can be reused

Results



Platform: Dell Precision T5400, 3.16GHz quad-core Xeon X5460 dual processor, 8GB
Benchmarks: NAS parallel benchmark suite v 2.3, Barcelona OpenMP task suite v 1.0
Compilers: ROSE, Omni 1.6, GCC 4.4.1, Mercurium 1.3.3 compiler with Nanos 4.1.4 runtime. Intel compiler 11.1.059

Early Experiences with the OpenMP Accelerator Model

Chunhua Liao, Yonghong Yan*, Bronis R. de Supinski, Daniel J. Quinlan, and Barbara Chapman*

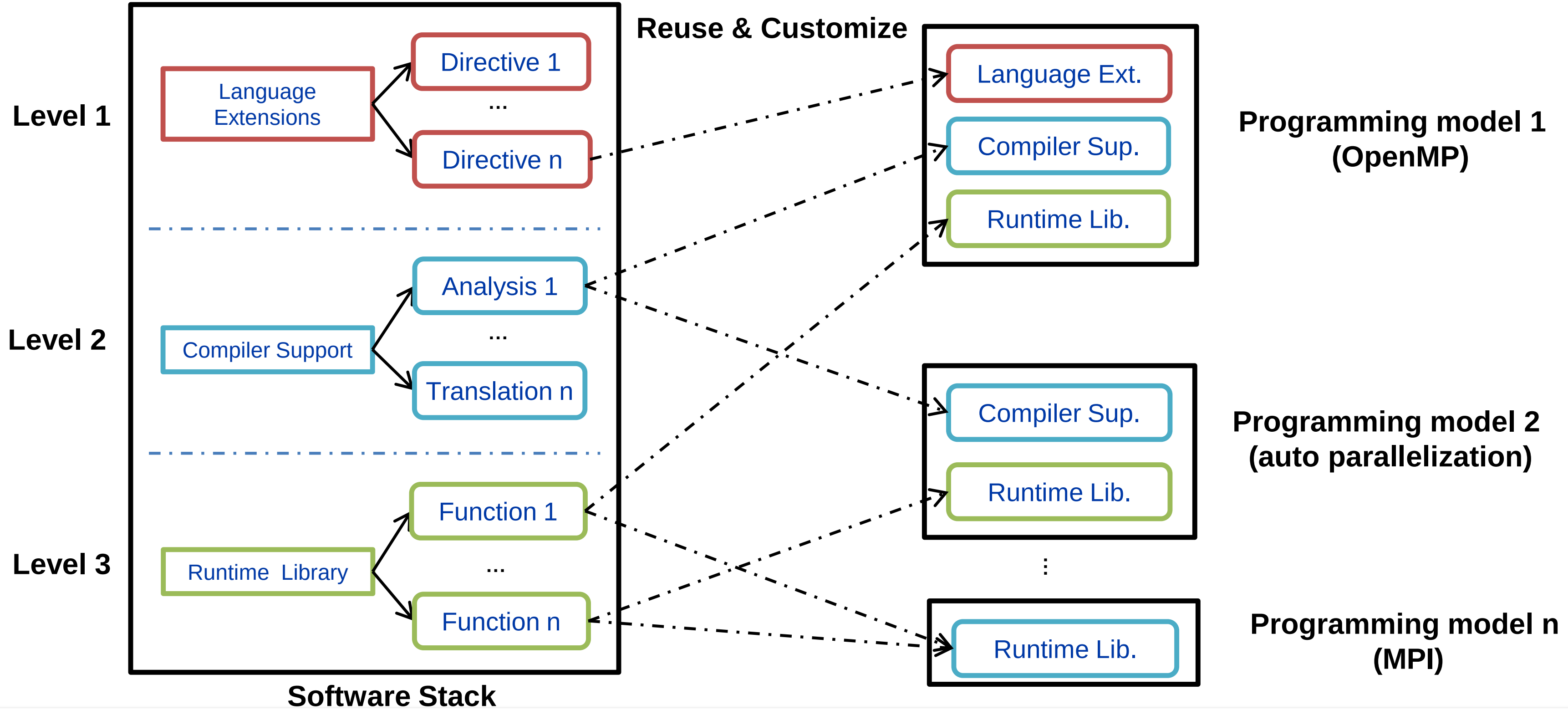
Lawrence Livermore National Laboratory

* University of Houston

LLNL-PRES- 642558

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Building Blocks in Software Stack and Different Types of Programming Models



Example Programming Model: Heterogeneous OpenMP (HOMP)

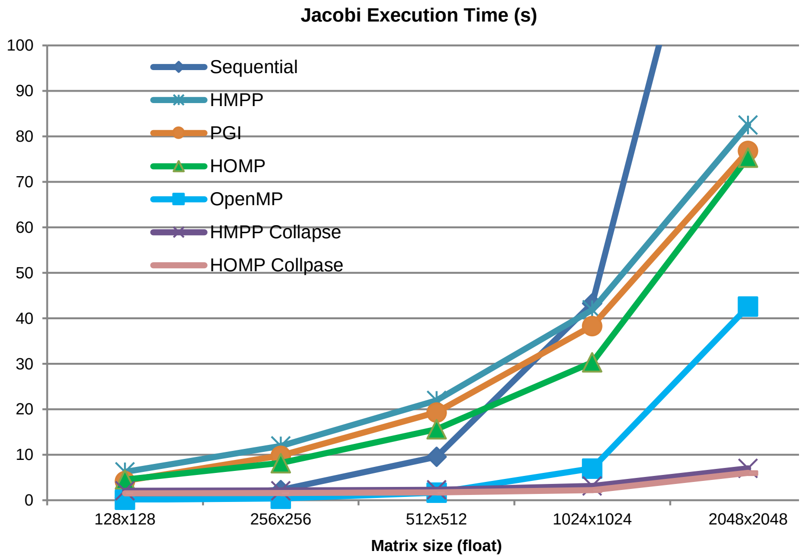
- Goal
  - Address heterogeneity challenge of exascale: computing using accelerators
  - Discover/develop building blocks for directive parsing, compiler translation, and runtime support
- Approach
  - Explore extensions to a general-purpose programming model
    - OpenMP accelerator model: OpenMP directives + accelerator directives
  - Build on top of existing OpenMP implementation in ROSE

Language level:
target, device, map,
num_devices, reduction,
device_type, no-mid-sync, …

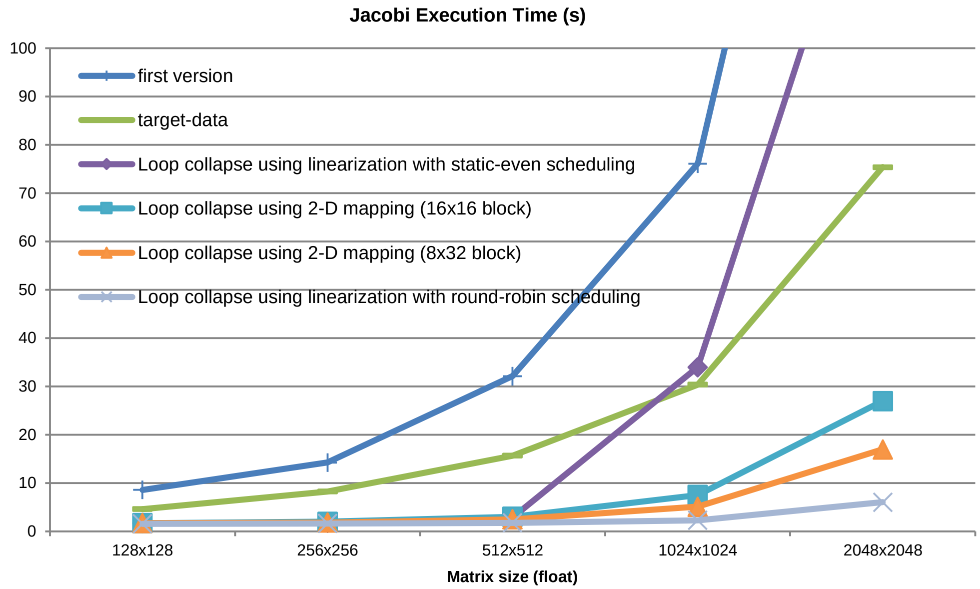
Compiler Level:
parser building blocks
outliner, loop transformation,
data transformation, …

Runtime Level:
device probing,
data management, loop
scheduling, reduction, …

Results



Jacobi Execution Time (s)
100
90
80
70
60
50
40
30
20
10
0
Sequential
HMPP
PGI
HOMP
OpenMP
HMPP Collapse
HOMP Collpase
128x128
256x256
512x512
1024x1024
2048x2048
Matrix size (float)



Jacobi Execution Time (s)
100
90
80
70
60
50
40
30
20
10
0
first version
target-data
Loop collapse using linearization with static-even scheduling
Loop collapse using 2-D mapping (16x16 block)
Loop collapse using 2-D mapping (8x32 block)
Loop collapse using linearization with round-robin scheduling
128x128
256x256
512x512
1024x1024
2048x2048
Matrix size (float)

Agenda

- OpenMP Compilers
- Tools
- Benchmarks
- Metrics
- Ongoing and Future work
- Conclusion

The ROSE AST Outliner

Outlining: semantically the reverse transformation of inlining

- Form a function from a specified code segment
- Replace the code segment with a call to the function

Widely used in many scenarios

- Test case generation
- Implementation of OpenMP for CPUs and GPUs
- Just-in-time compilation
- Autotuning of whole programs

A More Effective Outliner

- Collect code segments (outlining targets) via interface
- Perform side-effect and liveness analysis
- Bottom up traverse the AST and process each target
  - Check the eligibility of a target
  - Create an outlined function
    - Create a function skeleton with parameters
    - Handle function parameters: decide pass by value vs. reference
    - Move the target into the outlined function's body
    - Replace variable references: variable cloning to avoid pointer uses
  - Replace the target with a call to the outlined function

```
Usage: outline [OPTION]... FILENAME...
Main operation mode:
        -rose:outline:preproc-only
        -rose:outline:abstract_handle handle_string
        -rose:outline:parameter_wrapper
        -rose:outline:structure_wrapper
        -rose:outline:enable_classic
        -rose:outline:temp_variable
        -rose:outline:enable_liveness
        -rose:outline:new_file
        -rose:outline:output_path
        -rose:outline:exclude_headers
        -rose:outline:use_dlopen
        -rose:outline:enable_debug
```

outline -rose:outline:abstract_handle ”ForStatement<position,12>” -rose:outline:use_dlopen test3.cpp
// outline the for loop located at line 12 of test3.cpp, call it using dlopen

Algorithm Details

Parameter Handling: reduce parameters

- Scope and linkage
  - C: global only
  - C++: global vs. class-scope , C-linkage
- Parameters: for control and data
  - Goal: a few parameters as possible
  - Rely on scope, side effect and liveness analysis

1. Parameters = ((AllVars − InnerVars − GlobalVars − NamespaceVars −ClassVars) ∩ (LiveInVars U LiveOutVars)) U ClassPointers

2. PassByRefParameters = Parameters ∩ ((ModifiedVars ∩ LiveOutVars) U ArrayVars U ClassVars)

Reducing Pointer Dereferences

- We use a novel method: variable cloning
  - Check if such a variable is used by address: address-taken analysis
    - C: &x;
    - C++: T & y=x; or foo(x) when foo(T&)
  - Use a clone variable if x is NOT used by address

3. CloneCandidates = PassByRefParameters ∩ PointerDereferencedVars

4. CloneVars = (CloneCandidates − UseByAddressVars) ∩ AssignableVars

5. CloneVarsToInit = CloneVars ∩ LiveInVars

6. CloneVarsToSave = CloneVars ∩ LiveOutVars

Classic vs. Effective Outlining

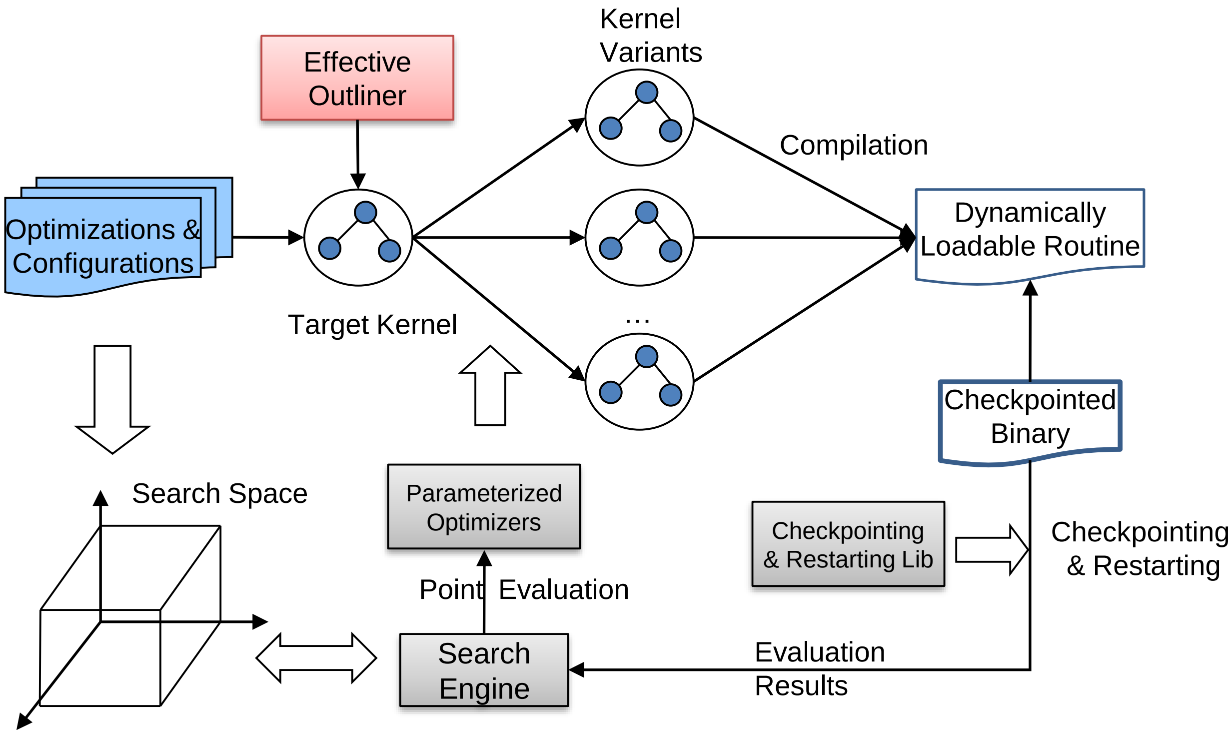
Classic algorithm with pointer-dereferencing

```
void OUT__1__4027__(int *ip__, int *jp__, double omega, double
*errorp__, double *residp__, double ax, double ay, double b)
{



// Four variables become pointers: i,j, resid, error
  for (*ip__=1;*ip__<(n-1);(*ip__)++)
    for (*jp__=1;*jp__<(m-1);(*jp__)++)
    {
      *residp__ = (ax * (uold[*ip__-1][*jp__] + uold[*ip__+1][*jp__]) +
           ay * (uold[*ip__][*jp__-1] + uold[*ip__][*jp__+1]) +
           b * uold[*ip__][*jp__] - f[*ip__][*jp__])/b;
      u[*ip__][*jp__] = uold[*ip__][*jp__] - omega * (*residp__);
      *errorp__ = *errorp__ + (*residp__) * (*residp__);
    }
}
```

Effective Outlining

```
void OUT__1__5058__(double omega,double *errorp__, double
ax, double ay, double b)
{
  int i, j;      /* neither live-in nor live-out*/
  double resid ; /* neither live-in nor live-out */
  double error ; /* clone for a live-in and live-out parameter */
  error = *errorp__; /* Initialize the clone*/
  for (i = 1; i < (n - 1); i++)
    for (j = 1; j < (m - 1); j++) {
      resid = (ax * (uold[i - 1][j] + uold[i + 1][j]) +
          ay * (uold[i][j - 1] + uold[i][j + 1]) +
          b * uold[i][j] - f[i][j]) / b;
      u[i][j] = uold[i][j] - omega * resid;
      error = error + resid * resid;
    }

  *errorp__ = error; /* Save value of the clone*/
}
```

The Outliner Used for Whole Program Autotuning



SMG (semicoarsing multigrid solver) 2000

- 28k line C code, stencil computation
- 120x120x129 data set
- a kernel ~45% execution time for

Results:

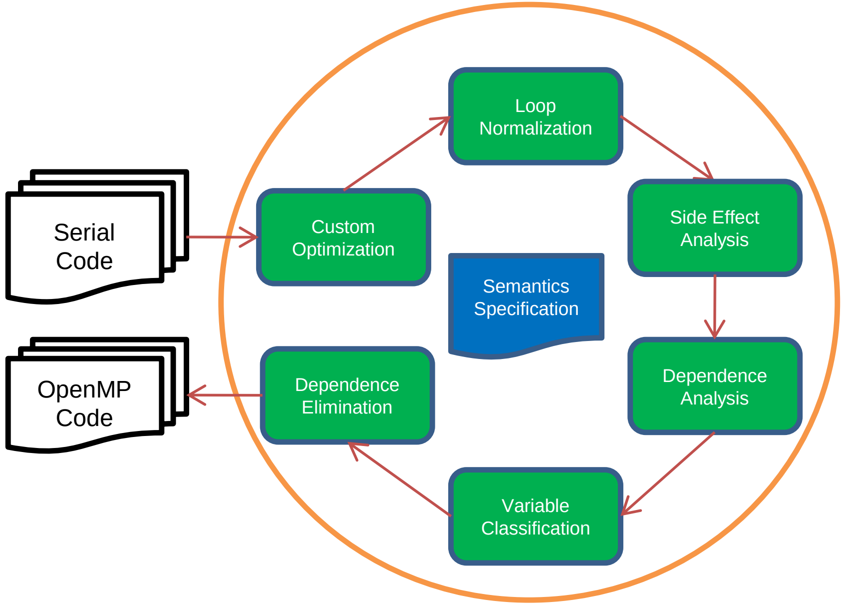
- 5.55x Speedup for kernel
- 1.76x Speedup for total execution time

The Need for Better C++ Parallelization Tools

- Extreme-scale architectures: abundant parallelism provided by heterogeneous components (such as multicore CPUs + GPUs)
  - Existing C++ HPC applications exploit only coarse-grain parallelism via MPI

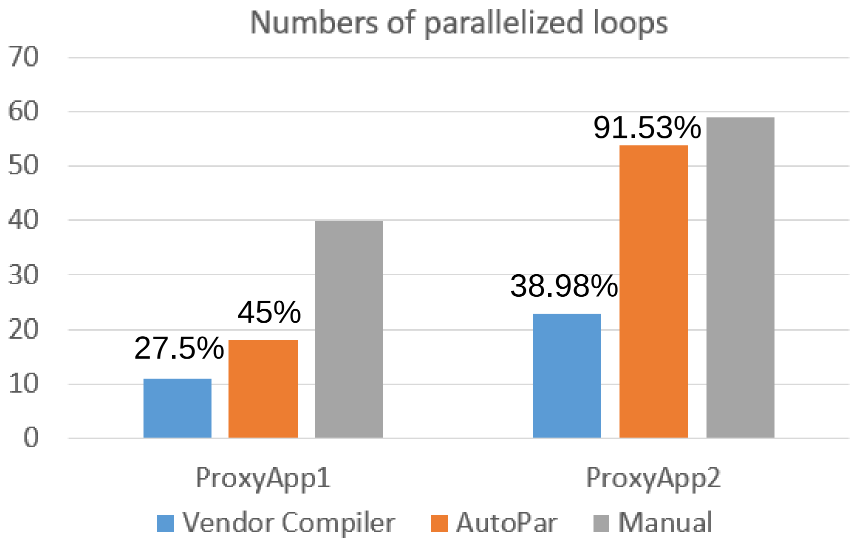
- Existing parallelization tools mostly focus on Fortran or C applications.
  - Depend on conventional compilers using low level internal representation (IR)
    - Difficult to discover high-level abstractions
  - Even more challenging to extract/leverage associated semantics

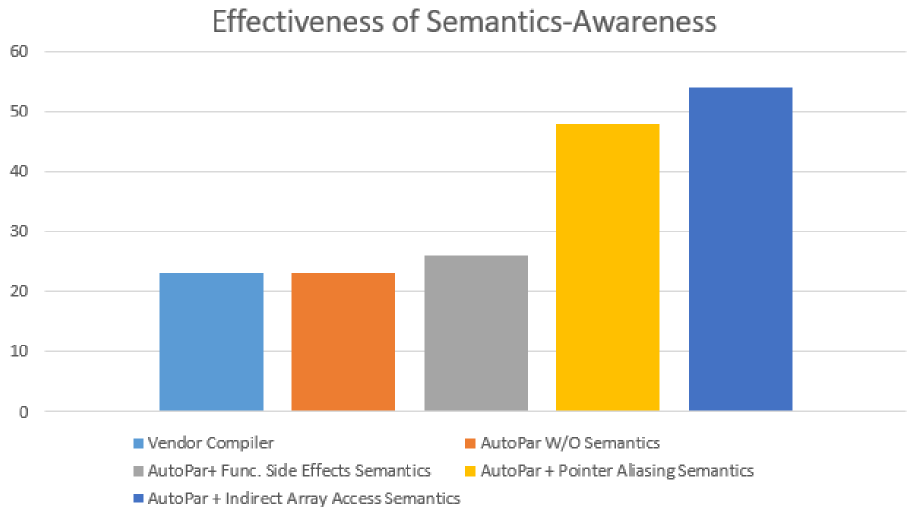
AutoPar: Semantics-Aware Automatic Parallelization




```
class  floatArray {     // user defined array abstraction
  alias none; overlap none; //elements are alias-free and non-overlapping
  is_fixed_sized_array { //semantic-preserving functions as a fixed-sized array
    length(i) = {this.size()};  // array semantics: obtain length
    element(i) = {this.operator[](i); this.elem(i);}; // array element access
semantics
  };
};

std::list<SgFunctionDef*> findCFunctionDefinition(SgNode* root){
  read {root};  modify {result}; //side effects of a function
  return unique; //return a unique set
}

operator pow(double val1, double val2)
   {
    modify none; read {val1, val2}; alias none;
   }
```

Results





ProxyApp2

Additional Features Requested by LLNL Application Teams

- Undo loop normalization: users want their loops unchanged.
- Generate patches instead of outputting files with scattered changes
- Support checking correctness of existing OpenMP directives
- Verify correctness of generated OpenMP codes
  - Connecting to third-party tools like Intel Inspector to catch data races
  - User-provided semantics can be wrong
  - Code can have bugs which lead to data races: found a big one! → spawn another tool
- Reduce the number of private variables for globally scoped variables
  - Move variable declarations to inner scopes→ spawn another tool

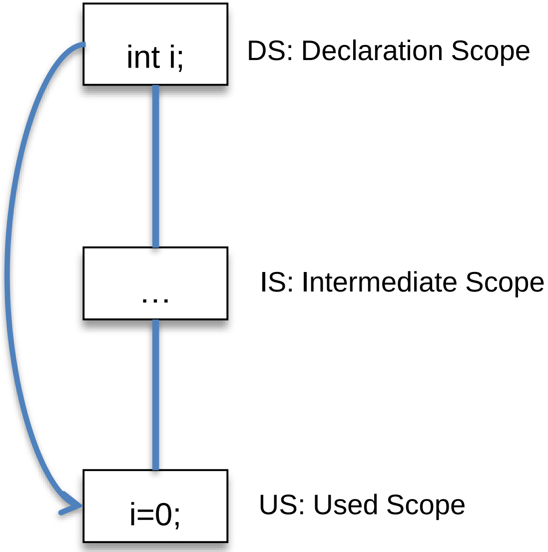
The Move Tool: a Code Refactoring Tool to Move Variable Declarations into Innermost Scopes

- A source-to-source refactoring tool to support ASC application teams
  - Copy-move variable declarations into innermost scopes: variable privatization
  - Benefits: facilitate code parallelization (migrating to OpenMP/RAJA)
- Algorithm went through 3 versions
  - V1: Naïve single-round move
  - V2: Iterative move using a declaration worklist
  - V3: Separated analysis and move: much more efficient

Case 1: Single Used Scope vs. Case 2: Multiple Used Scopes

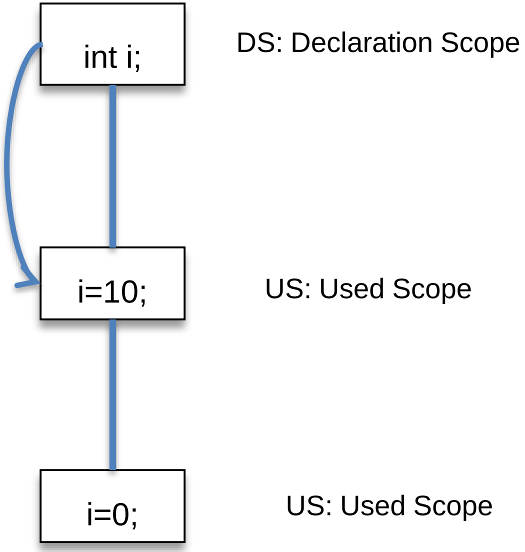
```
void foo()
{
 int i;
 {
   {
     i =0;
   }
 }
}
```

Code with a declaration



a scope tree:
three types of Scope Nodes
parent-child edges

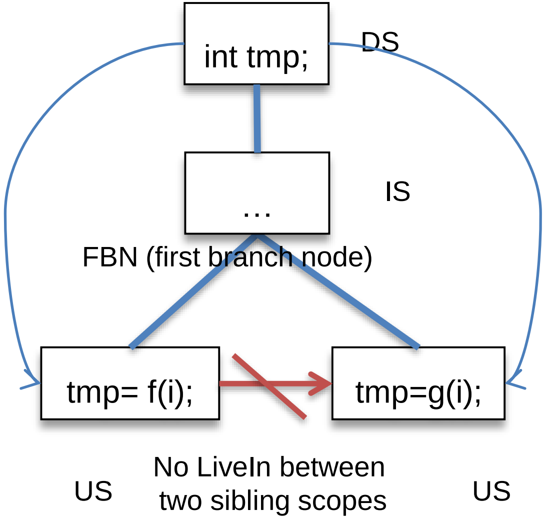
```
void foo()
{
 int i;
 {
   i = 10;
   {
     i =0;
   }
 }
}
```



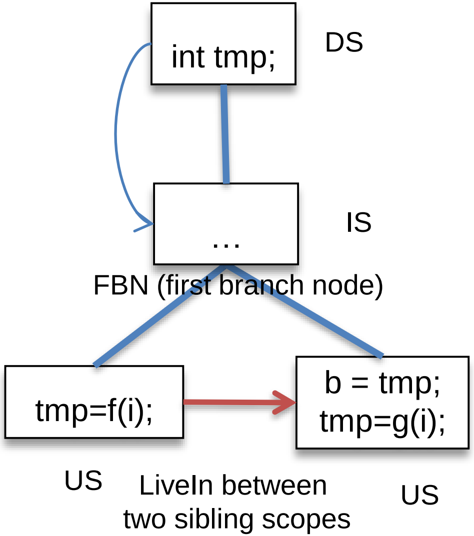
scope tree with multiple used scopes
* trim shadowed used scope

Case 3: Multiple Used Scope Branches of the Same Length

```
{
  int tmp ;
  {
   {
      tmp = f(i) ;
    }
    /*… */
    {
     tmp = g(i) ;
    }
  }
}
```



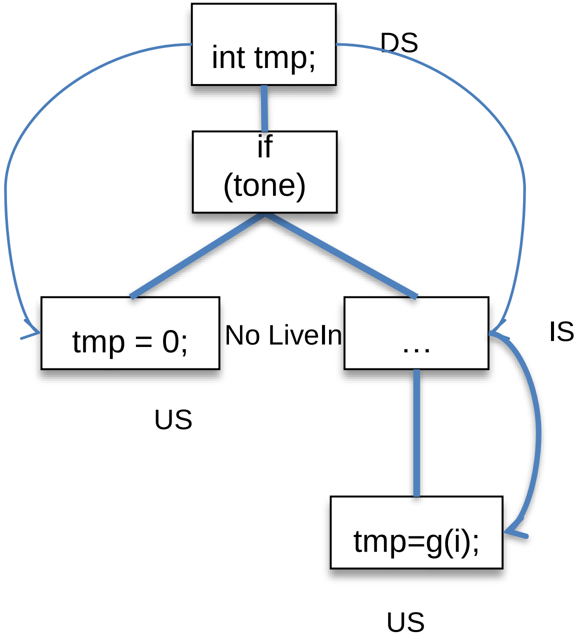

```
{
  int tmp ;
  {
   {
      tmp = f(i) ;
    }
    /*  … */
    {
      b = tmp;
      tmp = g(i) ;
    }
  }
}
```



Baseline algorithm V1: handles case 1,2 and 3

Case 4: Multiple Branches with Different Lengths

```
int  tmp;
if (tone)
{
     tmp = 0;
}
else
{
  {
     {
       tmp = 0;
     }
  }
}
```

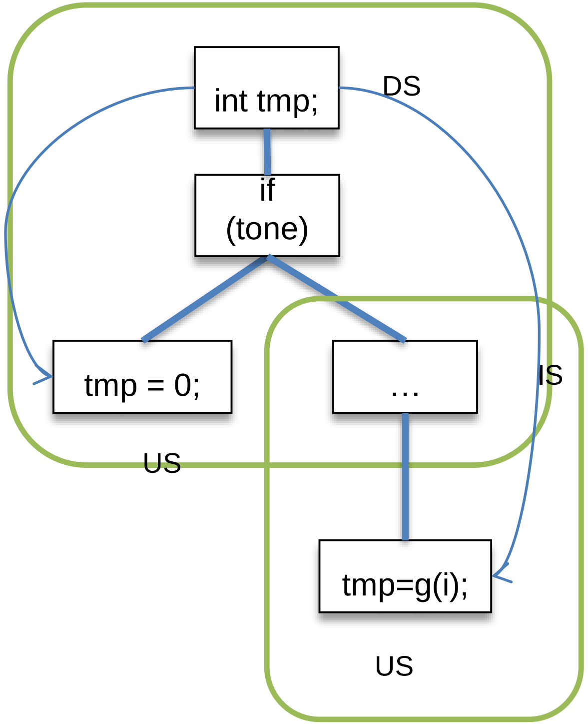


Need further moves

Algorithm V2: iteratively move declarations

- A declaration copy-moved to a new location
  - the newly inserted declaration should be considered for further movements
  - Focus on declarations
- An iterative algorithm using a worklist
  - initial worklist = original declarations in the function
  - while (!worklist.empty())
    - decl = worklist.front(); worklist.pop();
    - moveDeclarationToInnermostScope(decl, inserted_decls);
    - worklist.push_back(each of inserted_decls)

Only Need to Find Final Scopes and Move Once: Algorithm V3



- Find final scopes first
  - scope_tree_worklist.push(scope_tree);
  - while (!scope_tree_worklist.empty())
    - current_scope_tree = scope_tree_worklist.front(); …
    - collectCandidateTargetScopes(decl, current_scope_tree);
      - if (is a bottom scope?)
        - target_scopes.push_back(candidate)
      - else
        - scope_tree_worklist.push_back(candiate)
- Then copy&move in one shot
  - if (target_scopes.size()>0)
    - copyMoveVariableDeclaration(decl, target_scopes);

Results

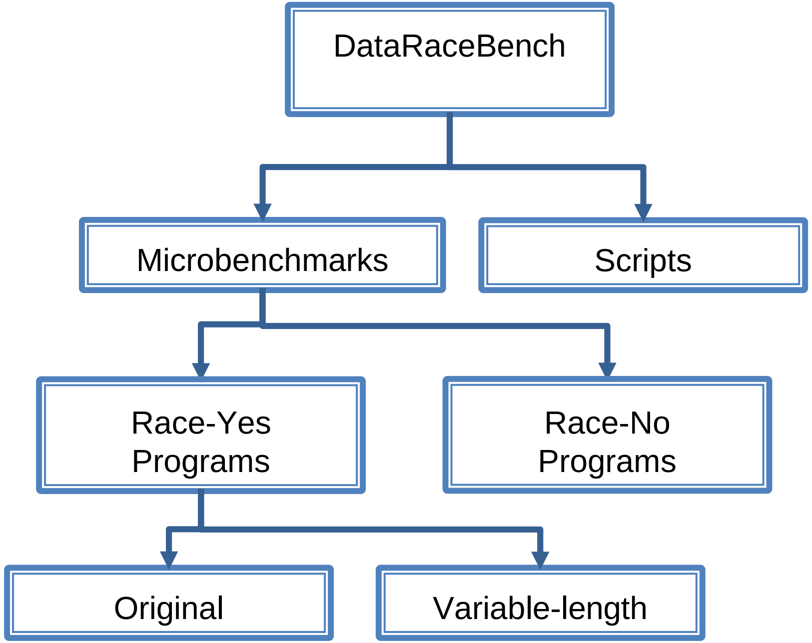
- 230+ regression tests, many with correctness verification (diff-based)
- Applied to large-scale X,Y apps, very positive user feedback
- Users kept requesting more features once previous requests were met
  - merge moved declarations with immediately followed assignments
  - transformation tracking, debugging support
  - aggressive mode, keep-going mode, no-op mode, …

Agenda

- OpenMP Compilers
- Tools
- Benchmarks
- Metrics
- Ongoing and Future work
- Conclusion

DataRaceBench: a Dedicated Benchmark Suite to Evaluate Data Race Detection Tools

- Motivation
  - the lack of apple-to-apple comparison among tools
- Coverage: 116 total microbenchmarks
  - V1.0.1: 72 from AutoPar’s regression tests, PolyOpt, LLNL apps, etc.
  - v1.2.0: 44 more based on semantics coverage analysis of OpenMP 4.5



https://github.com/LLNL/dataracebench

Design Philosophy: Both Positive and Negative Tests

```
1. …
2. int i,x;
3. #pragma omp parallel for
4.  for (i=0;i<100;i++)
5.  {   x=i;  }
6.  printf("x=%d",x);
7. …
```

lastprivatemissing-orig-yes.c

one data race pair
x@5 vs. x@5
Y2: Missing data sharing clauses

```
1. …
2. int i,x;
3. #pragma omp parallel for lastprivate (x)
4. for (i=0;i<100;i++)
5.  {   x=i; }
6. printf("x=%d",x);
7. …
```

lastprivate-orig-no.c

N2: Use of data sharing clauses

Example Numerical Kernel with Data Races

```
1.    int indexSet[180] = {
2.    521, 523, 525, 527, 529, 533,
3.    547, 549, 551, 553, 555, 557,...
4.     };
5.    double * xa1, *xa2; ...
6.    xa2 = xa1 + 12 ;
7.    #pragma omp parallel for
8.    for(int i=0; i< 180; ++i)
9.    {
10.     int idx=indexSet[i];
11.     xa1[idx]+=1.0;
12.     xa2[idx]+=3.0;
13.   }
```

indirectaccess2-orig-yes.c
Data race pair:
xa1[idx]@11 vs. xa2[idx]@12

Loop carried data dependence

iteration i	**0**	**1**	**…**	**5**
indexSet[i]	521	523	…	533
xa1[indexSet[i]]	base+521		…	base+533
xa2[indexSet[i]]	base+12+521		…	

Test Sensitivity to Thread Count & Loop Scheduling

Loop iterations

0	1	2	3	4	5	…	177	178	179

Loop schedule with 2 threads

0	…	5	…	89

thread 1/2

90	91	..	178	179

thread 2/2

Loop schedule 2 with 36 threads

0	…	4

thread 1/36

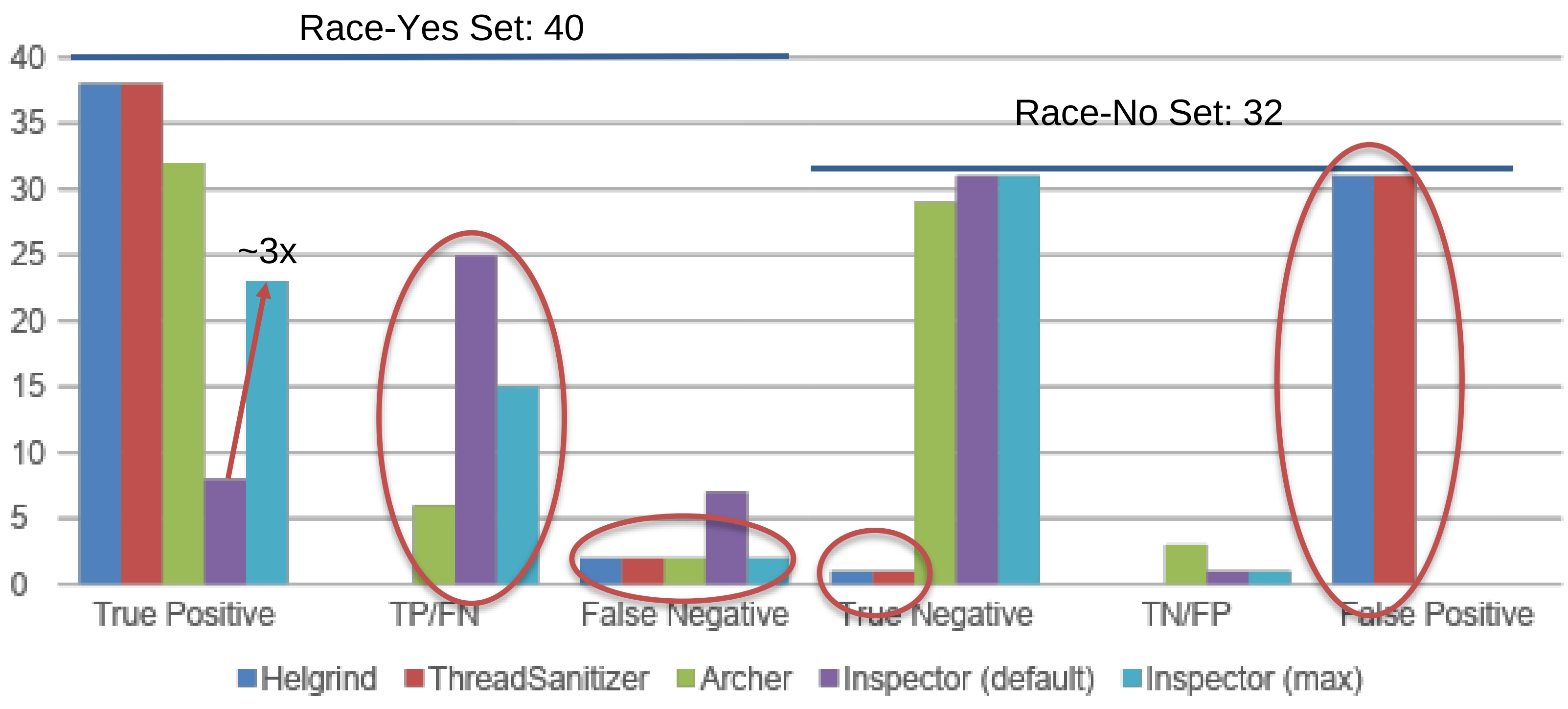
5	…	9

thread 2/36

…

180/36=5

V1.0.1 Results: Positive and Negative Counts



V1.2.0 Results for Archer and Intel Inspector

Microbenchmark Program	R	Archer min - max race	Archer type	Intel Inspector min - max race	Intel Inspector type
DRB073-doall2-orig-yes.c	Y	84 - 92	TP	2 - 2	TP
DRB074-flush-orig-yes.c	Y	1 - 3	TP	1 - 1	TP
DRB075-getthreadnum-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB076-flush-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB077-single-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB078-taskdep2-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB079-taskdep3-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB080-func-arg-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB081-func-arg-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB082-declared-in-func-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB083-declared-in-func-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB084-threadprivatemissing-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB085-threadprivate-orig-no.c	N	-	CSF	0 - 0	TN
DRB086-static-data-member-orig-yes.cpp	Y	-	CSF	1 - 1	TP
DRB087-static-data-member2-orig-yes.cpp	Y	-	CSF	1 - 1	TP
DRB088-dynamic-storage-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB089-dynamic-storage2-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB090-static-local-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB091-threadprivate2-orig-no.c	N	-	CSF	0 - 0	TN
DRB092-threadprivatemissing2-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB093-doall2-collapse-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB094-doall2-ordered-orig-no.c	N	-	CUN	0 - 0	RTO

Microbenchmark Program	R	Archer min - max race	Archer type	Intel Inspector min - max race	Intel Inspector type
DRB095-doall2-taskloop-orig-yes.c	Y	-	CUN	2 - 2	TP
DRB096-doall2-taskloop-collapse-orig-no.c	N	-	CUN	0 - 4	FP TN
DRB097-target-teams-distribute-orig-no.c	N	0 - 0	RSF	0 - 0	TN
DRB098-simd2-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB099-targetparallelfor2-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB100-task-reference-orig-no.cpp	N	-	CUN	0 - 0	TN
DRB101-task-value-orig-no.cpp	N	0 - 0	TN	0 - 0	TN
DRB102-copyprivate-orig-no.c	N	-	CSF	0 - 0	TN
DRB103-master-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB104-nowait-barrier-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB105-taskwait-orig-no.c	N	0 - 0	TN	3 - 4	FP
DRB106-taskwaitmissing-orig-yes.c	Y	35 - 48	RTO TP	4 - 6	TP
DRB107-taskgroup-orig-no.c	N	0 - 0	TN	1 - 1	FP
DRB108-atomic-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB109-orderedmissing-orig-yes.c	Y	71 - 71	TP	1 - 1	TP
DRB110-ordered-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB111-linearmissing-orig-yes.c	Y	73 - 85	TP	1 - 2	TP
DRB112-linear-orig-no.c	N	-	CUN	0 - 0	TN
DRB113-default-orig-no.c	N	0 - 0	TN	0 - 0	TN
DRB114-if-orig-yes.c	Y	42 - 48	TP	1 - 1	TP
DRB115-forsimd-orig-yes.c	Y	44 - 47	TP	1 - 1	TP
DRB116-target-teams-orig-yes.c	Y	0 - 0	RSF	1 - 1	TP

Compile-time seg. fault (CSF), Unsupported feature (CUN)

Runtime seg. Fault (RSF), Runtime timeout (RTO)

Conclusion

- Lessons about execution settings
  - Configurations of dynamic tools matter: Intel default vs. max resources
  - Multiple runs: necessary to increase probability of finding data races
  - Sensitive to the number of threads and scheduling policies
- Findings about results from v1.0.1
  - Precision/Accuracy: Archer and Intel Inspector win over Helgrind and ThreadSanitizer due to OpenMP awareness
  - User friendliness: Only Intel inspector consolidates multiple data race instances into one single pair of source locations
  - SIMD loops with data races: compilers do not generate SIMD instructions for our race-yes SIMD benchmarks
- Re-evaluated two tools using v1.2.0
  - Intel Inspector supported more microbenchmarks without compilation or runtime errors than Archer did
  - Room for improvements for Intel Inspector to support taskloop, taskwait, taskgroup, etc.

Rethinking the success metric of HPC

HPC = Highly Painful Computing

Sacrificing hours of hard human (graduate students) cycles for a few reduced machine cycles in research papers

Would some application teams really want to use the HPC software/hardware systems we dump on them every 3-5 years, if they had choices??

Success(HPC) = f(FLOPs, Watt)

A New Holistic Success Metric for HPC as a Service

Success(HPC) =

f (Total_time, Quality_of_results, Total_cost, Context)

- Total_time = the entire end-to-end, machine-human interaction time to get results
  - Human_time = training, thinking_steps, keystrokes, mouse_clicks, cursor_travel_distance, ~~hairs_pulled_off~~, …
- Quality_of_results:
  - Correctness, accuracy, certainty/confidence, up-to-date …
- Total_cost = Machine_cost + Human_cost
  - Human cost tied to hourly rates: make HPC operable/usable by even cavemen
- Context: under which conditions can HPC serve users (including cavemen)?
  - Access devices (smartphones), Locations (AOE), Time (24x7), …

HPC = Highly Pleasant Computing

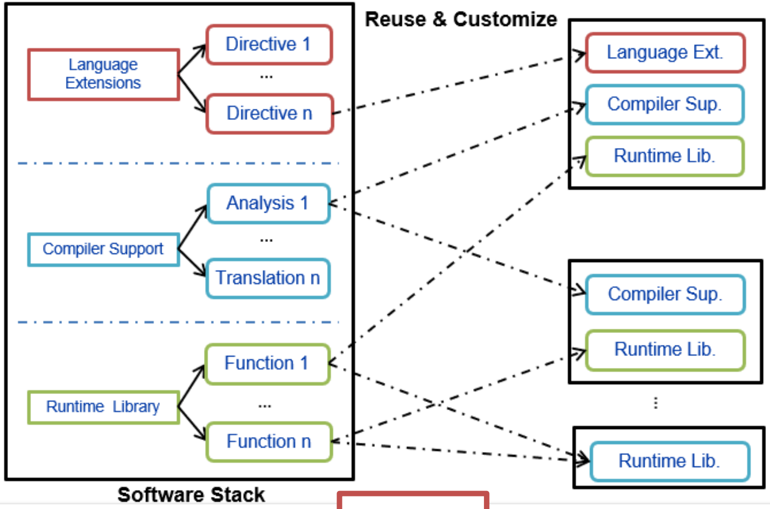
R&D Focus 1: Benchmarking as a Service

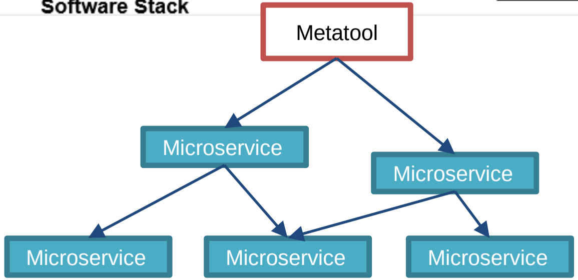
If You Can't Measure it Correctly, You Can't Improve it

- More
  - Regression positive/negative tests for all layers in the software stack, for each component e.g. fundamental program analyses and optimizations
- Better
  - Collaboratively define holistic metrics of success by developers and users
- Automated
  - 1 button to start the process and get all results: e.g. scripts+ docker for DataRaceBench
- Continuous
  - Auto-regenerate results if anything changes: benchmarks, tools, environments …
- Live results : https://github.com/LLNL/dataracebench/wiki/Tool-Evaluation-Dashboard
  - A scoreboard to show the state-of-the-art in objective, quantitative ways
  - For sponsors, users, and researchers

R&D Focus 2: Program Analysis and Optimization as Services

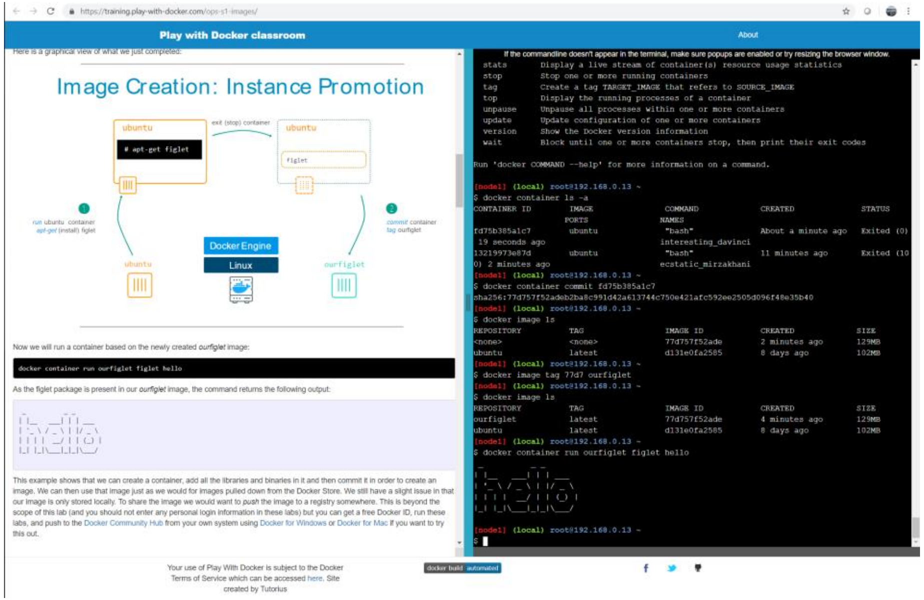
- Hard to use individual tools by using compile-> run
  - Make them into services available 24x7: through local or web instances, again possibly dockerized
  - Define standard APIs and data formats to accept input and give out output
- Multiple tools/services needed to address HPC challenges (e.g. programming models, automatic parallelization)
  - Making compilers and tools as interoperable, composable services
  - Enable building programming models and metatools by composing services





R&D Focus 3: Automatic, Adaptive Online Training/Certifying HPC Researchers & Developers

- Problem:
  - Many programming models (extensions) to explore
  - Many tools requested by app teams
  - But only limited FTEs available
- Solution: automatic training and certifying researchers & developers
  - Modern Learning Management Systems (LMS) + Adaptive Learning/Assessement
  - FreeCodeComp → FreeCompilerComp
  - Play-with-Docker → Play-with-HPC



Takeaway Messages

1. Programming models (e.g. OpenMP): fast prototyping requires interchangeable building blocks at language, compiler, and runtime layers
   a) Need common APIs, exchangeable data formats, provided as microservices
2. Tools: need right metrics to communicate incremental progress with users
   a) Regression tests = positive tests + negative tests
   b) Commenting on issues of apps = commenting on issues of children in front of their parents!
3. Benchmarks: people love and hate benchmarks
   a) Best qualified people may not want to develop/release the best benchmarks for their work
4. Success(HPC)=f (Flops, Watt) ➔ Highly Painful Computing for people
   a) Let's refine the metric to include human factors together and make it Highly Pleasant Computing
5. It is a new golden age for HPC researchers: largely overlooked human cycle optimization
   a) Benchmarking, reproducibility, microservice design, docker, cloud, adaptive online training/certification, ….

Acknowledgements

- Ph.D. Advisor: Barbara Chapman
- ROSE project lead: Dan Quinlan
- Co-workers/collaborators: Bronis, Markus Schordan, Xipeng Shen, Yonghong Yan, …
- Sponsors: DOE Office of Science -ASCR, LLNL-LDRD, ECP/ATDM, ARMY…
  - XPlacer: Extensible and Portable Optimizations of Data Placement in Memory (10/17-9/20),
  - RAPIDS, SciDAC Institute for Computer Science and Data (10/17-9/20),
  - Exascale Code Generation Toolkit (10/16-3/19),
  - Data Race Free HPC Software (10/16-9/19),
  - Institute for Sustained Performance, Energy, and Resilience (SUPER) (10/11-09/16),
  - DSL Technology for Exascale Computing (D-TEC) (09/12-08/15),
  - FAIL-SAFE: Fault Aware Intelligent Software For Exascale (12/13-07/15),
  - A Node-Level Programming Model Framework for Exascale(04/12-03/15),
  - Compiled MPI: Cost-Effective Exascale Application Development (10/10-09/13),
  - Thrifty: An Exascale Architecture for Energy Proportional Computing (10/10-09/13),
  - CoDEx: A Hardware/Software Codesign Environment for the Exascale Era (10/10-09/13),
  - Semantics-Driven Optimization of Scientific Applications (10/09-09/12),
  - …

Lawrence Livermore National Laboratory