Representing Parallelism Within LLVM – Can We Have Our Cake and Eat It Too?

Hal Finkel, Johannes Doerfert, Xinmin Tian (Intel), George Stelle (LANL)

EuroLLVM 2018

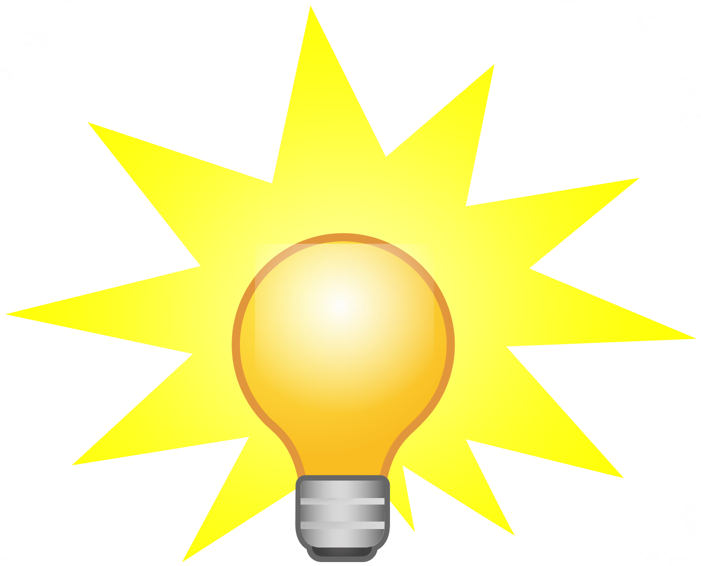
Why are we interested in parallelism-aware optimizations?

- To optimize code that exists today.
- To optimize code that exists in the near future.

Some code we see in the near future could look significantly different from the code we commonly see today. Why?

- Because of OpenMP for GPUs (and other accelerators).
- Because of upcoming OpenMP features.
- Because of C++ parallel-algorithms libraries.

New Features in OpenMP and the Optimizer...



Motivation

```
!! !$acc loop gang
!$omp teams distribute

!! !$acc loop gang worker
!$omp teams distribute parallel do

!! !$acc loop gang vector
!$omp teams distribute simd

!! !$acc loop gang worker vector
!$omp teams distribute parallel do simd

!! !$acc loop worker
!$omp parallel do

!! !$acc loop vector
!$omp simd

!! !$acc loop worker vector
!$omp parallel do simd
```

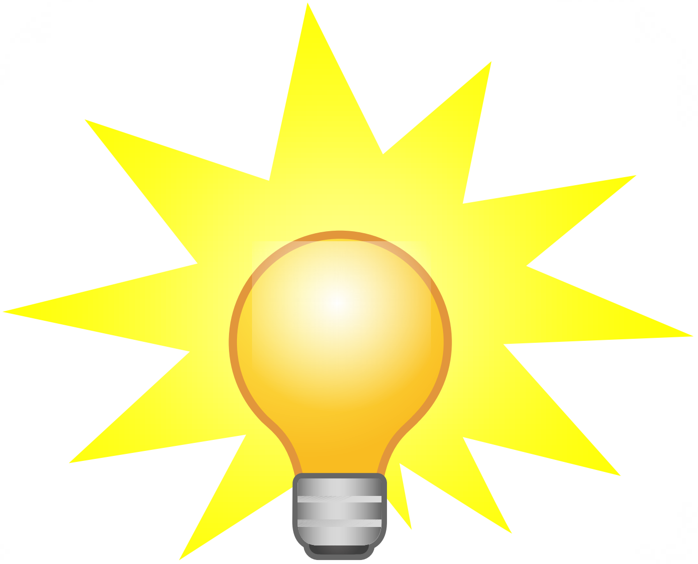
```
!$acc loop gang worker
do i=1,N
    !$acc loop independent
    do j=1,N
       ...
    end do
end do
```

- Initially proposed by NVidia, ORNL and LBL
- Education purpose
- Portable code with understand of non optimal performance

```
!$omp target teams distribute parallel do
 do i=1,N
    !$omp concurrent
    do j=1,N
       ...
    end do
end do
```

Compiler Backend Decision for Parallelization and Vectorization

C++ Parallel-Algorithms Libraries...



C++ Parallel-Algorithms Libraries

We used to see only coarse-grained OpenMP, but this is changing…

- We're seeing even greater adoption of OpenMP, but…
- Many applications are not using OpenMP directly. Abstraction libraries are gaining in popularity.

Often uses OpenMP and/or other compiler directives under the hood.

Use of C++ Lambdas.

TBB and Thrust.

- RAJA (https://github.com/LLNL/RAJA)

```
RAJA::ReduceSum<reduce_policy, double> piSum(0.0);

RAJA::forall<execute_policy>(begin, numBins, [=](int i) {
  double x = (double(i) + 0.5) / numBins;
  piSum += 4.0 / (1.0 + x * x);
});
```

- Kokkos (https://github.com/kokkos)

```
int sum = 0;
// The KOKKOS_LAMBDA macro replaces
// the capture-by-value clause [=].
// It also handles any other syntax
// needed for CUDA.
Kokkos::parallel_reduce (n, KOKKOS_LAMBDA (const int i,
                                           int& lsum) {
    lsum += i*i;
  }, sum);
```

C++ Parallel-Algorithms Libraries

And starting with C++17, the standard library has parallel algorithms too...

Table 2 — Table of parallel algorithms

adjacent_difference	adjacent_find	all_of	any_of
copy	copy_if	copy_n	count
count_if	equal	exclusive_scan	fill
fill_n	find	find_end	find_first_of
find_if	find_if_not	for_each	for_each_n
generate	generate_n	includes	inclusive_scan
inner_product	inplace_merge	is_heap	is_heap_until
is_partitioned	is_sorted	is_sorted_until	lexicographical_compare
max_element	merge	min_element	minmax_element
mismatch	move	none_of	nth_element
partial_sort	partial_sort_copy	partition	partition_copy
reduce	remove	remove_copy	remove_copy_if
remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate
rotate_copy	search	search_n	set_difference
set_intersection	set_symmetric_difference	set_union	sort
stable_partition	stable_sort	swap_ranges	transform
transform_exclusive_scan	transform_inclusive_scan	transform_reduce	uninitialized_copy
uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n	unique
unique_copy			

[*Note:* Not all algorithms in the Standard Library have counterparts in Table 2. — *end note*]

```
// For example:
std::sort(std::execution::par_unseq, vec.begin(), vec.end()); // parallel and vectorized
```

(Missing) Optimizations for Parallel Programs

Or, “Why parallel loops might slow down your code”

Problem 1: variable capturing

Input program:

```
int y = 1337;
#pragma omp parallel for
for (int i = 0; i < N; i++)
  g(y, i);
g(y, y);
```

Optimal program:

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  g(1337, i);
g(1337, 1337);
```

Clang output:

```
int y = 1337;
call fork_parallel(fn, &y);
g(y, y);
```

GCC output:

```
int y = 1337;
call fork_parallel(fn, &y);
g(1337, 1337);
```

Solution 1: variable privatization

Input program:

```
int y = 1337;
#pragma omp parallel for
for (int i = 0; i < N; i++)
  g(y, i);
g(y, y);
```

Optimized program:

```
int y = 1337; y_p = y;
#pragma omp parallel for
for (int i = 0; i < N; i++)
  g(y_p, i);
g(1337, 1337);
```

Clang output:

```
int y = 1337;
call fork_parallel(fn, &y);
g(y, y);
```

Clang output:

```
int y_p = 1337;
call fork_parallel(fn, &y_p);
g(1337, 1337);
```

Problem 2: alias information

```
void work(int i, int *In) {

 ...

}

void foo(int * restrict In) {
  #pragma omp parallel for
  for (int i = 0; i < N; i++)
   work(i, In);
}
```

```
void work(int i,
                int * restrict In) {
 ...
}

void foo(int * restrict In) {
  #pragma omp parallel for
  for (int i = 0; i < N; i++)
   work(i, In);
}
```

Problem 2: alias information (con't)

```
void work(int i, int *In) {

 #critical
 *In = *In + i;

 #barrier

 #critical
 *In = *In + i;
}
```

```
void work(int i,
          int * restrict In) {
 #critical
 *In = *In + i;

 #barrier

 #critical
 *In = *In + i;
}
```

Solution 2: alias information propagation

```
void work(int i, int *In) {

 #critical
 *In = *In + i;

 #barrier

 #critical
 *In = *In + i;
}
```

```
void work(int i,
          int * restrict In) {
 #critical
 *In = *In + i;

 #barrier (use In)

 #critical
 *In = *In + i;
}
```

Problem 3: (implicit) barriers

```
void copy(float* dst, float* src, int N) {
 #pragma omp parallel for
  for(int i = 0; i < N; i++)
    dst[i] = src[i];
}
```

```
void compute_step_factor(int nelr, float* vars,
                         float* areas, float* sf) {
#pragma omp parallel for
  for (int blk = 0; blk < nelr / block_length; ++blk) {
      ...
```

Problem 3: (implicit) barriers (con't)

```
for (int i = 0; i < iterations; i++) {
  copy(old_vars, vars, nelr * NVAR);
  compute_step_factor(nelr, vars, areas, sf);
  for (int j = 0; j < RK; j++) {
    compute_flux(nelr, ese, normals, vars, fluxes, ff_vars,
                 ff_m_x, ff_m_y, ff_m_z, ff_dnergy);
    time_step(j, nelr, old_vars, vars, sf, fluxes);
  }
}
```

Problem 3: (implicit) barriers (con't)

```
for (int i = 0; i < iterations; i++) {
  #pragma omp parallel for              // copy
  for (...) { /* write old_vars, read vars */ }
  #pragma omp parallel for              // compute_step_factor
  for (...) { /* write sf, read vars & area */ }
  for (int j = 0; j < RK; j++) {
    #pragma omp parallel for            // compute_flux
    for (...) { /* write fluxes, read vars & ... */ }
    ...
```

Solution 3: region expansion & barrier elimination

```
#pragma omp parallel
for (int i = 0; i < iterations; i++) {
  #pragma omp for nowait          // copy
  for (...) { /* write old_vars, read vars */ }
  #pragma omp for  nowait         // compute_step_factor
  for (...) { /* write sf, read vars & area */ }
  for (int j = 0; j < RK; j++) {
    #pragma omp for               // compute_flux
    for (...) { /* write fluxes, read vars & ... */ }
  ...
```

Parallel-IR Optimizations

- Explore optimizations on different Parallel-IR representations.
- We want to collect evidence for
  - **cost** (implementation & compile time),
  - **effectiveness** (runtime improvements),
  - **integration & reusability** (in the pipeline),
  - **limitations** (that are unreasonable to work around).

Parallel-IR Optimizations Passes

class **ArgumentPrivatizationPass**;

class **AttributePropagationPass**;

class **BarrierEliminationPass**;

class **RegionExpansionPass**;

class **ComputationPlacementPass**;

class **ParallelRegion** {

contains(...)
getThreadId()
getExtend()

getKind()
getBarriers(...)

visitInstructions(...)
visitBlocks(...)

getCommunicatedValues()
replaceValueWith(...)

create(...)
createBarrier(...)
flattenParallelism(...)

}

Parallel-IR Representations

class **ArgumentPrivatizationPass**;

class **AttributePropagationPass**;

class **BarrierEliminationPass**;

class **RegionExpansionPass**;

class **ComputationPlacementPass**;

```
class ParallelRegion {
        contains(...)
        getThreadId()
        getExtend()
        getKind()
        getBarriers(...)
        visitInstructions(...)
        visitBlocks(...)

        getCommunicatedValues()
        replaceValueWith(...)

        create(...)
        createBarrier(...)
        flattenParallelism(...)
}
```

class **KMPCImpl** : ParallelRegion;

class **GOMPImpl** : ParallelRegion;

class **IntelPIRImpl**: ParallelRegion;

class **TapirImpl** : ParallelRegion;

class **LLVMPIRImpl** : ParallelRegion;

Parallel-IR Representations

class **ArgumentPrivatizationPass**;

class **AttributePropagationPass**;

class **BarrierEliminationPass**;

class **RegionExpansionPass**;

class **ComputationPlacementPass**;

class **ParallelRegion** {

contains(...)
getThreadId()
getExtend()

getKind()
getBarriers(...)

visitInstructions(...)
visitBlocks(...)

getCommunicatedValues()
replaceValueWith(...)

create(...)
createBarrier(...)
flattenParallelism(...)

}

class **KMPCImpl** : ParallelRegion;

class **GOMPImpl** : ParallelRegion;

class **IntelPIRImpl**: ParallelRegion;

class **TapirImpl** : ParallelRegion;

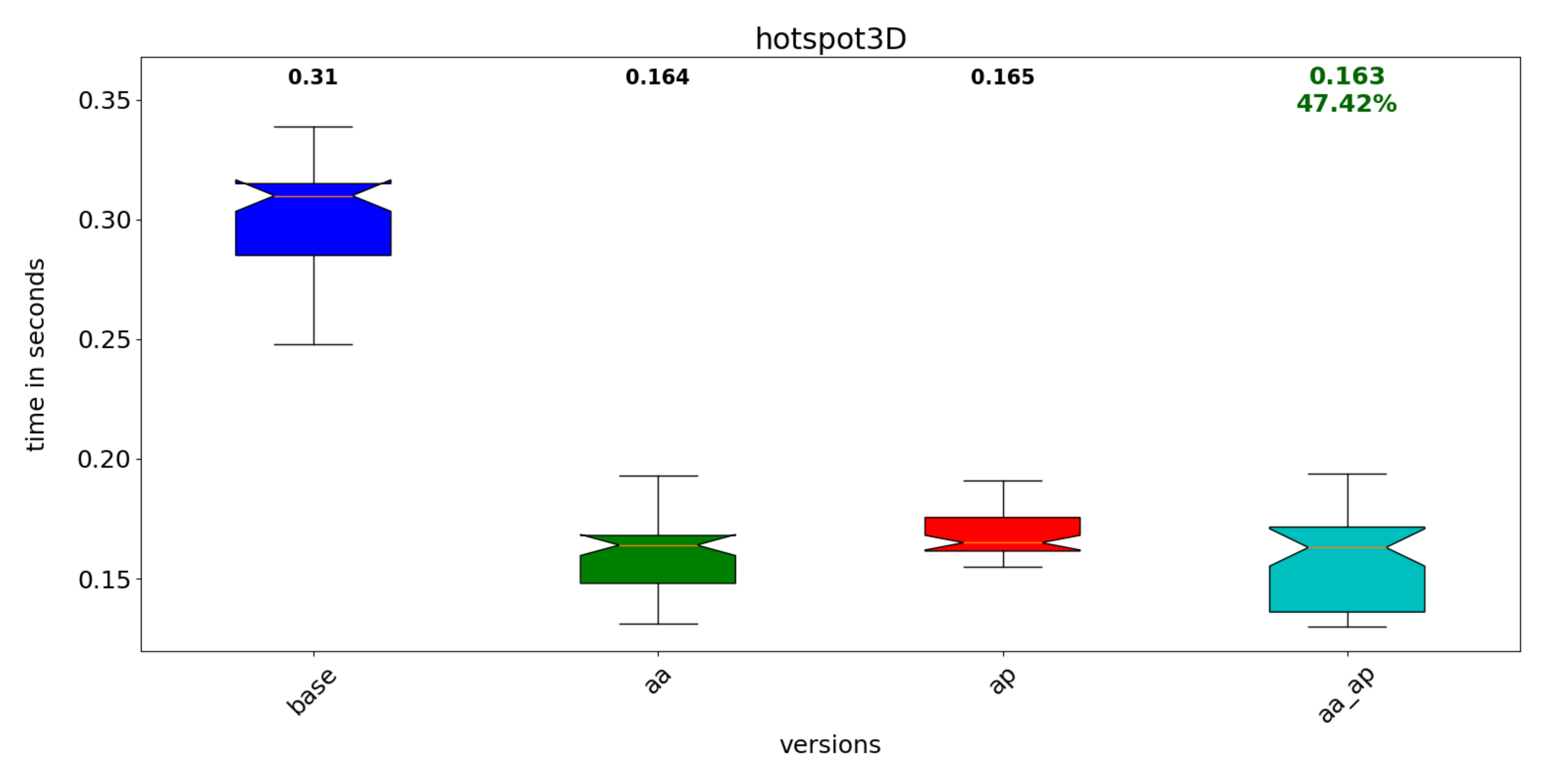
class **LLVMPIRImpl** : ParallelRegion;

Example 1: Rodinia - hotspot3D

```
#pragma omp parallel
{
  int count = 0;
  float *tIn = In, *tOut = Out;
#pragma omp master
  printf("%d threads running \n", omp_get_num_threads ());
  do {
    int z;
#pragma omp for
    for (z = 0; z < nz; z++) {
      int y;
      for (y = 0; y < ny; y++) {
        int x;
        for (x = 0; x < nx; x++) {
          int c, w, e, n, s, b, t;
          c = x + y * nx + z * nx * ny;
          w = (x == 0) ? c : c - 1;
          e = (x == nx - 1) ? c : c + 1;
          n = (y == 0) ? c : c - nx;
          s = (y == ny - 1) ? c : c + nx;
          b = (z == 0) ? c : c - nx * ny;
          t = (z == nz - 1) ? c : c + nx * ny;
          tOut[c] = cc * tIn[c] + cw * tIn[w] + ce * tIn[e] +
                    cs * tIn[s] + cn * tIn[n] + cb * tIn[b] +
                    ct * tIn[t] + (dt/Cap) * pIn[c] + ct * a;
        }
      }
    }
    float *t = tIn, tIn = tOut;
    tOut = t;
  } while (++count < numiter);
}
```

Example 1: Rodinia - hotspot3D

./3D 512 8 100 ../data/hotspot3D/power_512x8 ../data/hotspot3D/temp_512x8



Intel core i9, 10 cores, 20 threads, 51 runs, with and without

- aa => alias attribute propagation
- ap => argument privatization

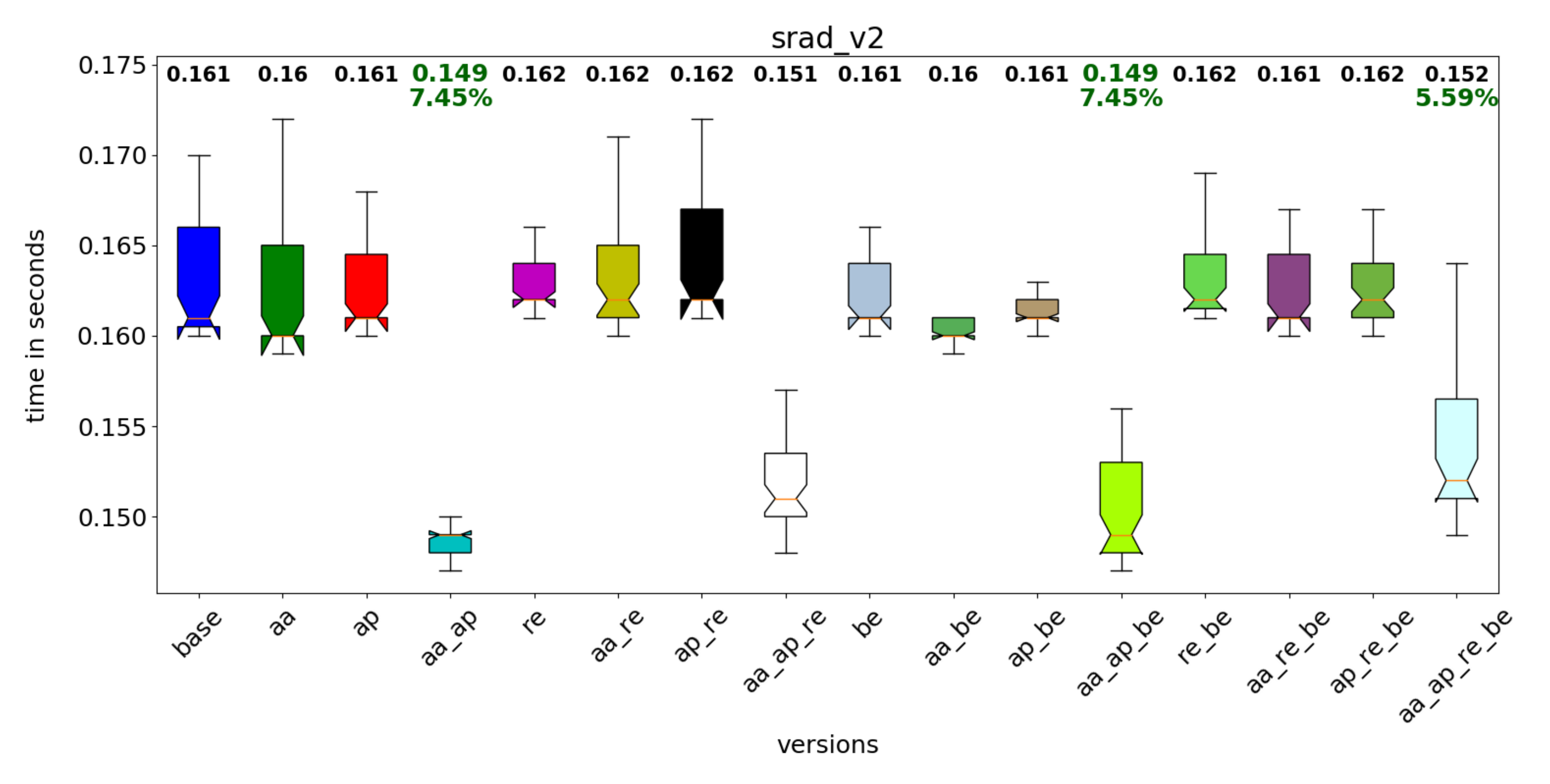
Example 2: Rodinia - srad_v2

```
#pragma omp parallel for shared(J, dN, dS, dW, dE, c, rows,    \
  cols, iN, iS, jW, jE) private(j, k, Jc, G2, L, num, den, qsqr)
for (int i = 0; i < rows; i++) {
  ...
}
#pragma omp parallel for shared(J, c, rows, cols, lambda)
                        private(i, j, k, D, cS, cN, cW, cE)
for (int i = 0; i < rows; i++) {
  ...
}
```

Example 2:

Rodinia - srad_v2

./srad 2048 2048 0 127 0 127 20 0.5 20



Intel core i9, 10 cores, 20 threads, 51 runs, with and without

- aa => alias attribute propagation
- ap => argument privatization
- re => region expansion
- be => barrier elimination

Example 3: Rodinia - cfd

```
for (int i = 0; i < iterations; i++) {
  copy(old_vars, vars, nelr * NVAR);
  compute_step_factor(nelr, vars, areas, sf);
  for (int j = 0; j < RK; j++) {
    compute_flux(nelr, ese, normals, vars, fluxes, ff_vars,
                  ff_m_x, ff_m_y, ff_m_z, ff_dnergy);
    time_step(j, nelr, old_vars, vars, sf, fluxes);
  }
}
```

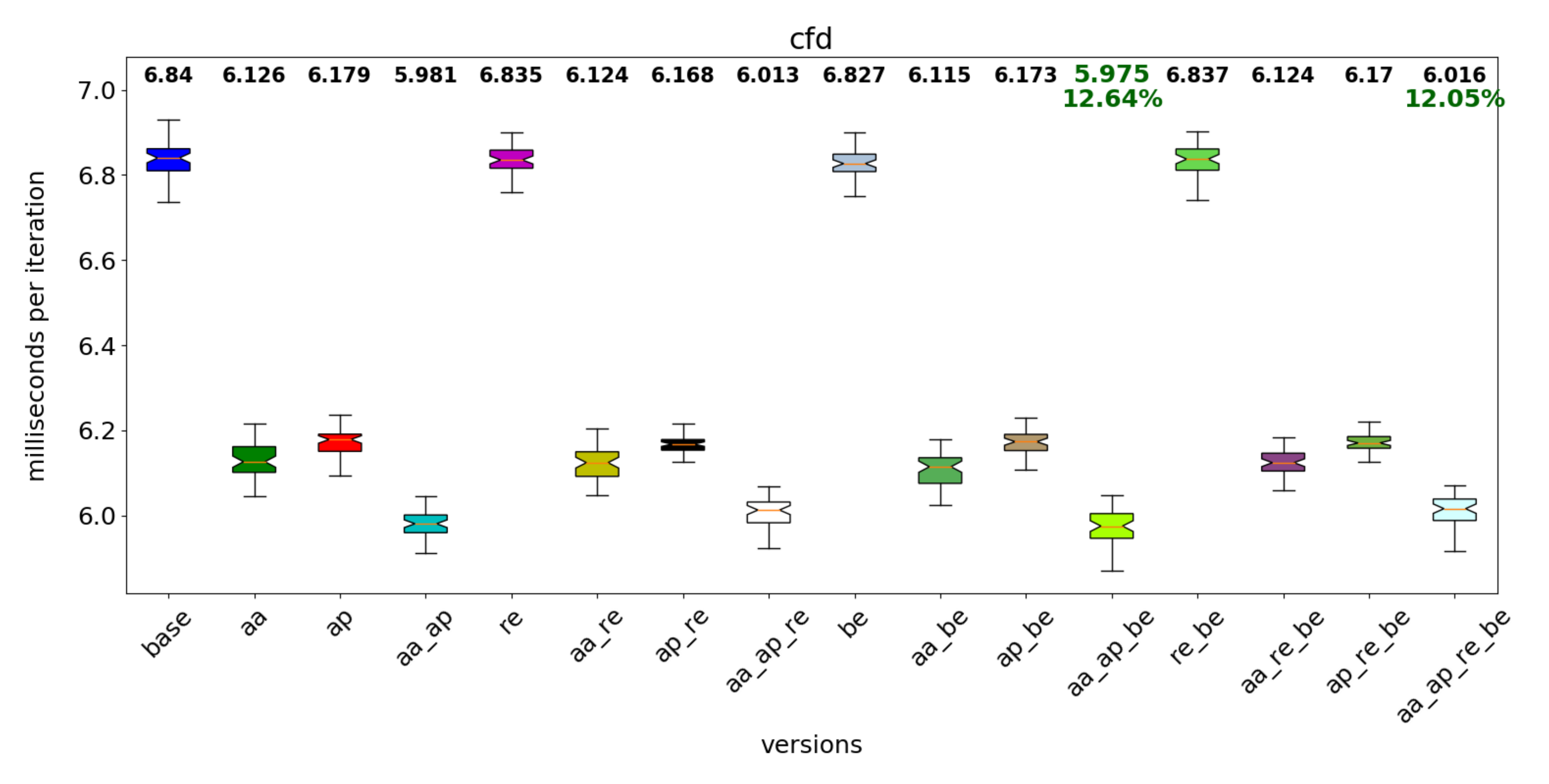
Example 3: Rodinia - cfd

```
#pragma omp parallel
for (int i = 0; i < iterations; i++) {
  #pragma omp for nowait              // copy
  for (...) { /* write old_vars, read vars */ }
  #pragma omp for nowait              // compute_step_factor
  for (...) { /* write sf, read vars & area */ }
  for (int j = 0; j < RK; j++) {
    #pragma omp for                   // compute_flux
    for (...) { /* write fluxes, read vars & ... */ }
  ...
```

Example 3:

Rodinia - cfd

cfd fvcorr.domn.193K

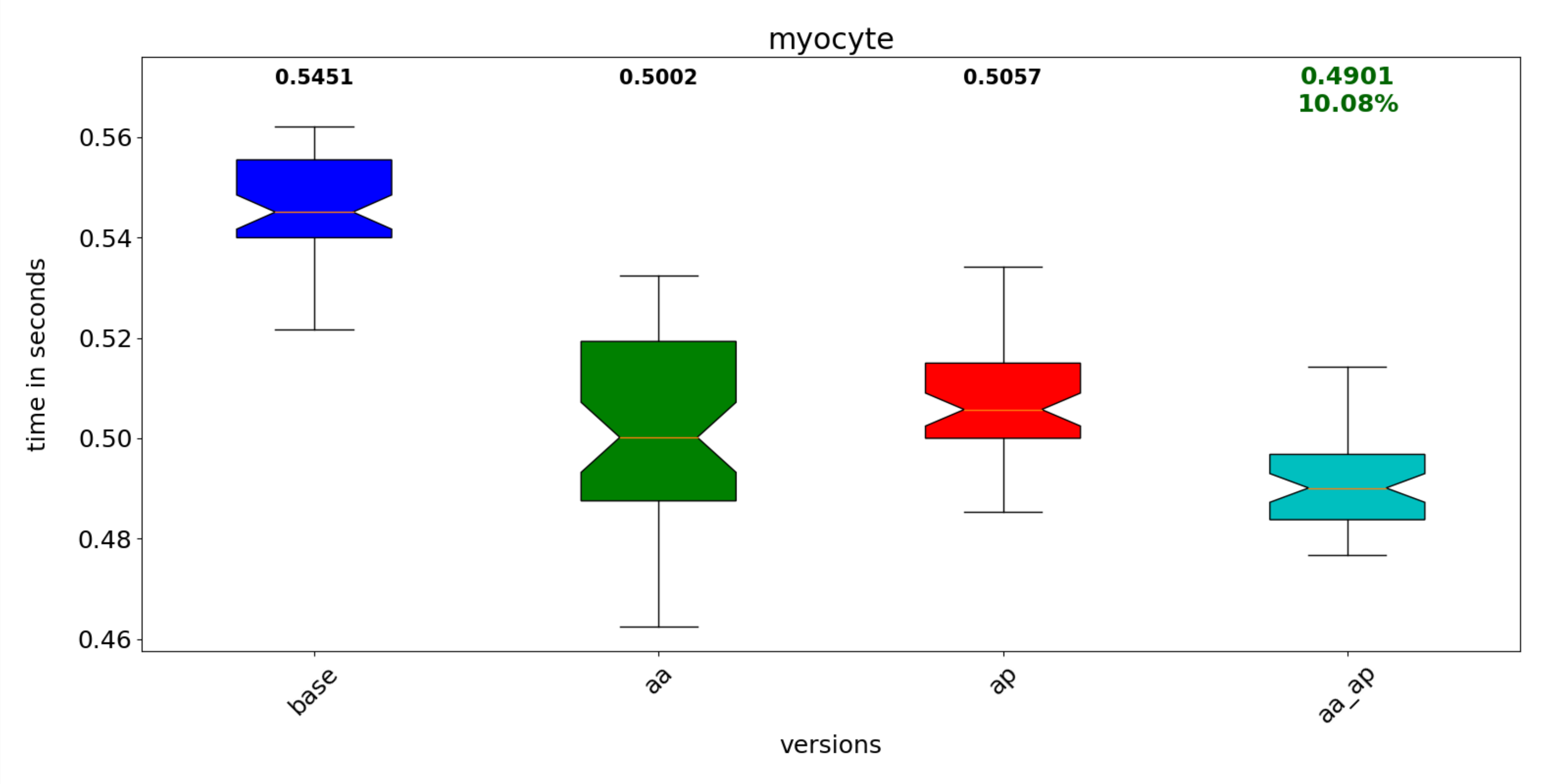


Intel core i9, 10 cores, 20 threads, 51 runs, with and without

- aa => alias attribute propagation
- ap => argument privatization
- re => region expansion
- be => barrier elimination

Example 4: Rodinia - myocyte

./myocyte 100 100 0 8



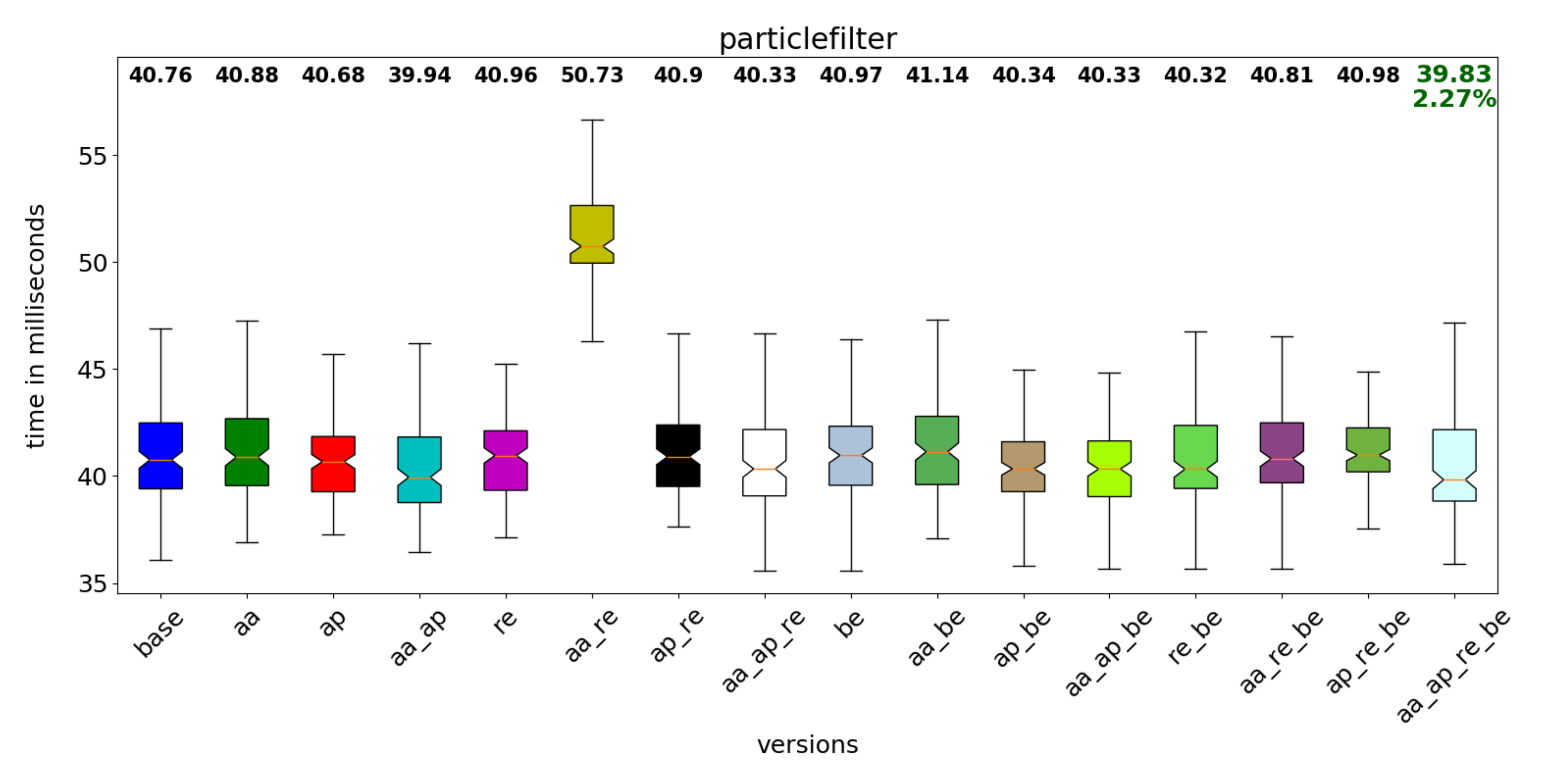
Intel core i9, 10 cores, 20 threads, 51 runs, with and without

- aa => alias attribute propagation
- ap => argument privatization
- re => region expansion
- be => barrier elimination

Example 5:

Rodinia - particlefilter

./particlefilter -x 128 -y 128 -z 10 -np 10000

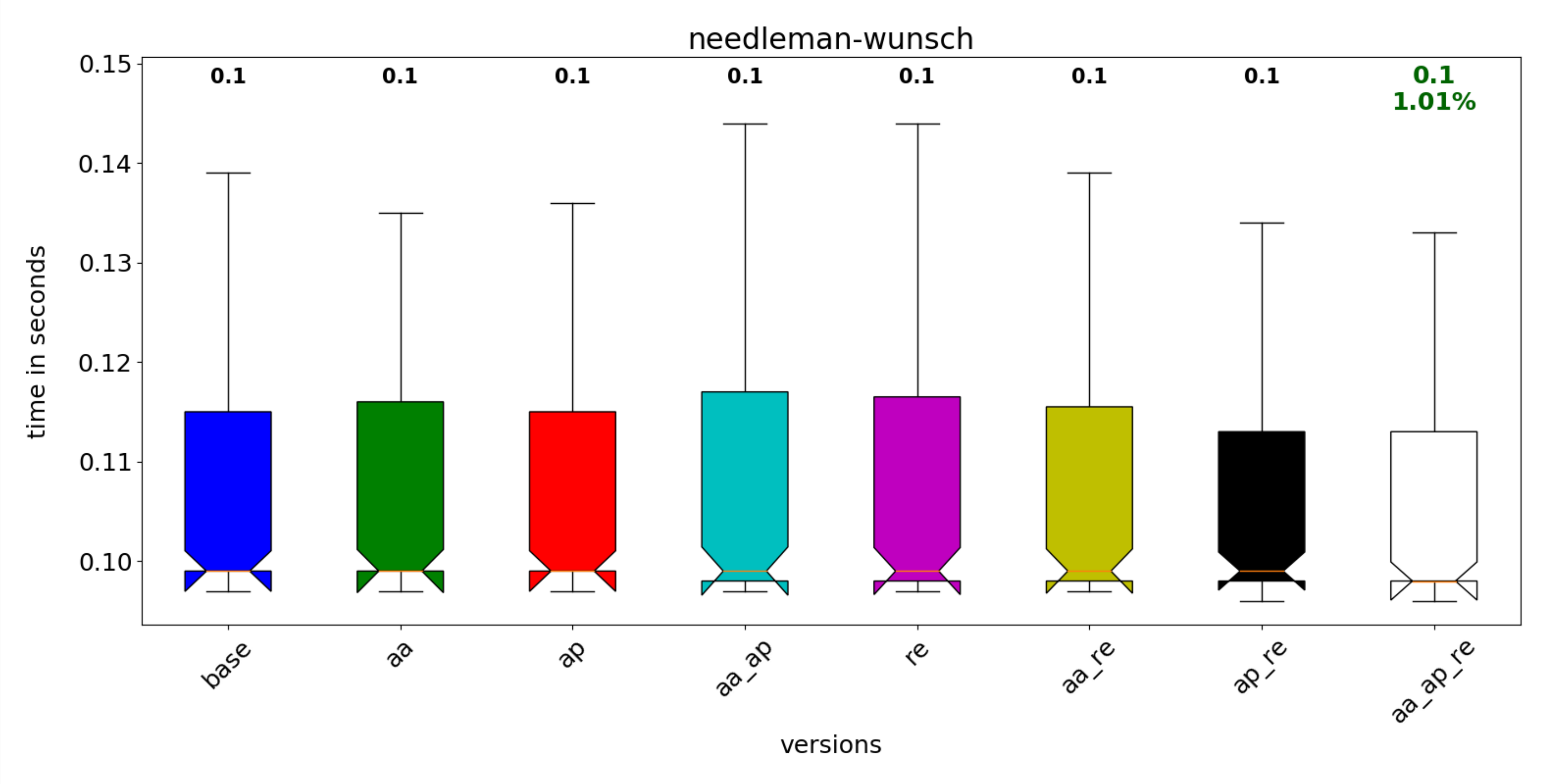


Intel core i9, 10 cores, 20 threads, 151 runs, with and without

- aa => alias attribute propagation
- ap => argument privatization
- re => region expansion
- be => barrier elimination

Example 6: Rodinia - needleman-wunsch

./nw 8192 10 8



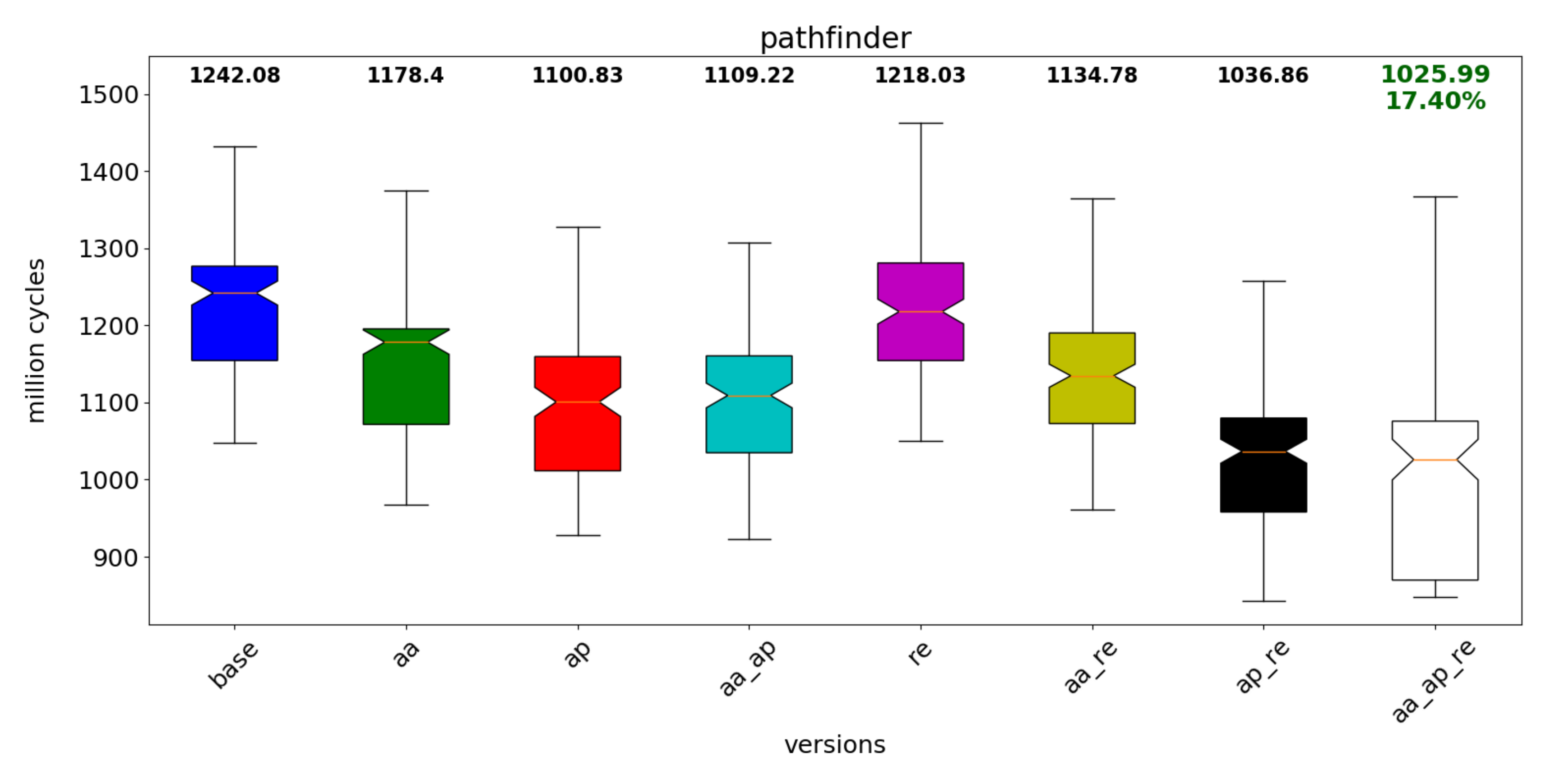
Intel core i9, 10 cores, 20 threads, 151 runs, with and without

- aa => alias attribute propagation
- ap => argument privatization
- re => region expansion
- be => barrier elimination

Example 7:

Rodinia - pathfinder

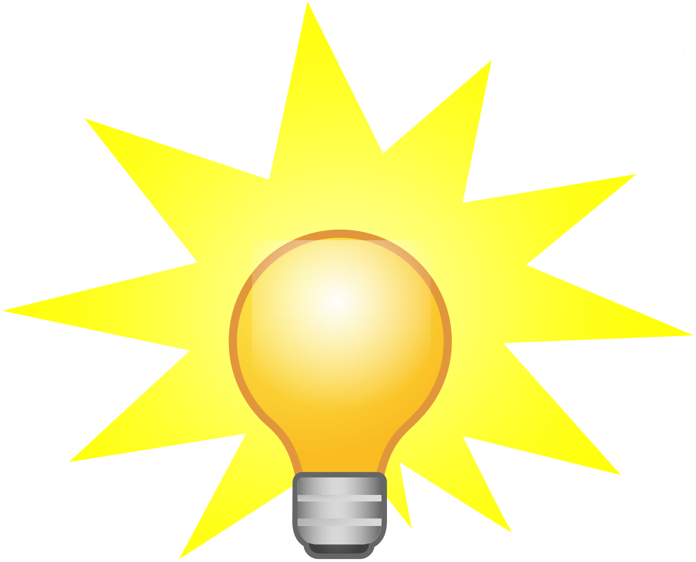
./pathfinder 40000 40000



Intel core i9, 10 cores, 20 threads, 151 runs, with and without

- aa => alias attribute propagation
- ap => argument privatization
- re => region expansion
- be => barrier elimination

What Intel Has Been Working On...



IR-Region Annotation RFC State (Intel and ANL)

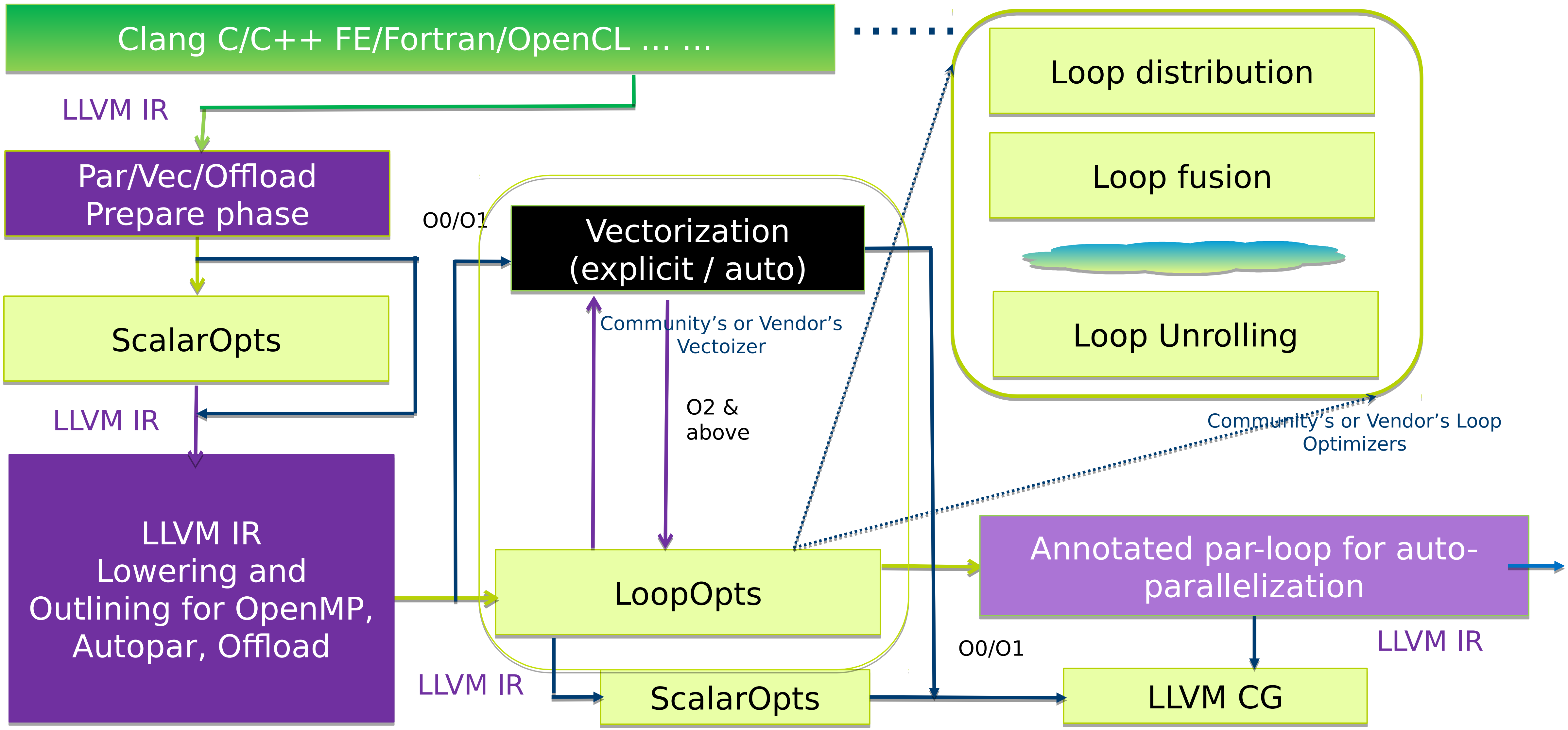
- Updated language agnostic LLVM IR extensions based on LLVM Token and OperandBundle representation (based on feedback from Google and Xilinx).
  - `def int_directive_region_entry : Intrinsic<[llvm_token_ty],[], []>;`
  - `def int_directive_region_exit : Intrinsic<[], [llvm_token_ty], []>;`
  - `def int_directive_marker : Intrinsic<[llvm_token_ty],[], []>;`
- Implemented explicit parallelization, SIMD vectorization and offloading in the LLVM middle-end based on IR-Region annotation for C/C++.
- Leveraged the new parallelizer and vectorizer for OpenCL explicit parallelization and vectorization extensions to build autonomous driving workloads.

IR-Region Annotation Usage Examples

```
#pragma omp target device(1) if(a) \
                   map(tofrom: x, y[5:100:1])
  structured-block

%t0 = call token @llvm.directive.region.entry()
                              [”DIR.OMP.TARGET”(),
                                             “QUAL.OMP.DEVICE”(1),
                                           “QUAL.OMP.IF”(type @a),
                                  “QUAL.OMP.MAP.TOFROM”(type *%x),
                  “QUAL.OMP.MAP.TOFROM:ARRSECT”(type *%y,1,5,100,1)]

      structured-block
call void @llvm.directive.region.exit(token %t0)
                          [”DIR.OMP.END.TARGET”()]
```

- **Parallel region/loop/sections**
- **Simd / declare simd**
- **Task / taskloop**
- **Offloading: Target map(…)**
- **Single, master, critical, atomics**
- **… …**

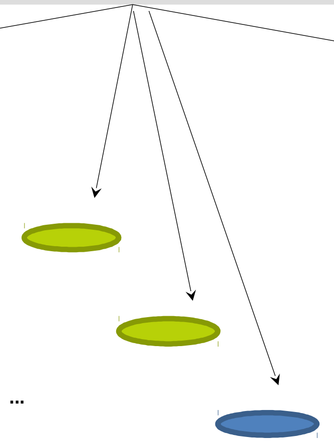
10000ft View: Intel® LLVM Compiler Architecture



W-Region Implementation

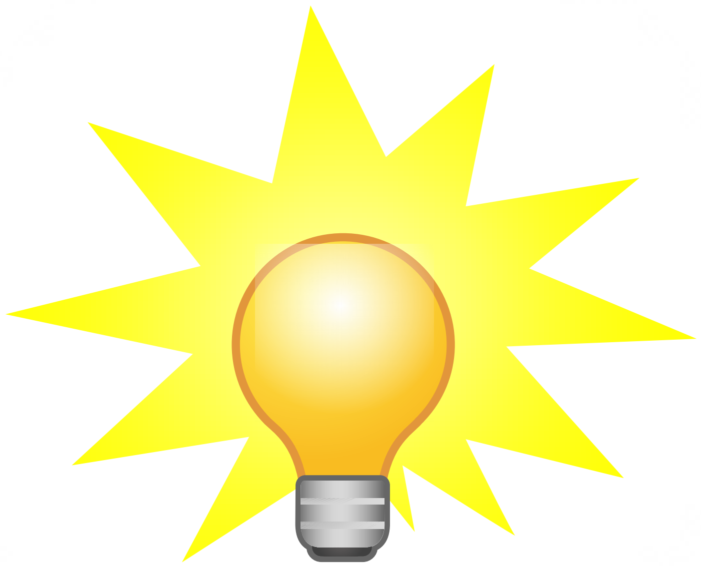
```
class WRN {  //base class
  BasicBlock *EntryBBlock;
  BasicBlock *ExitBBlock;
  unsigned nestingLevel;
  SmallVector<WRegionNode*,4> Children;
  …
}
```

```
// #pragma omp parallel
class Parallel : public WRN {
  SharedClause *Shared;
  PrivateClause *Private;
  Value NumThreads;
  …
}
```




```
// #pragma omp simd
class Simd : public WRN {
  PrivateClause *Private;
  LinearClause *Linear;
  int Simdlen;
  …
}
```

What LANL (+MIT, et al.) Has Been Working On...

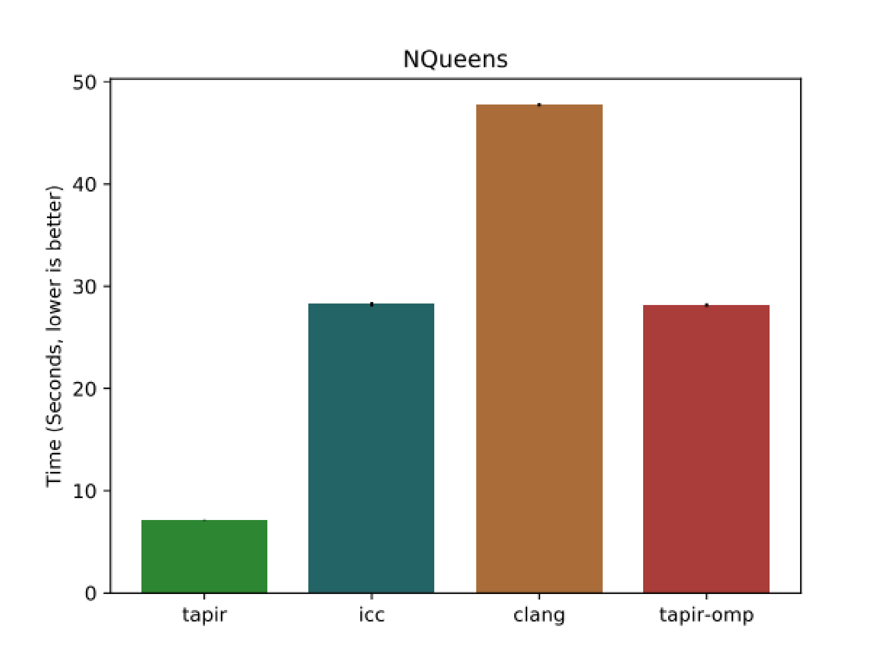


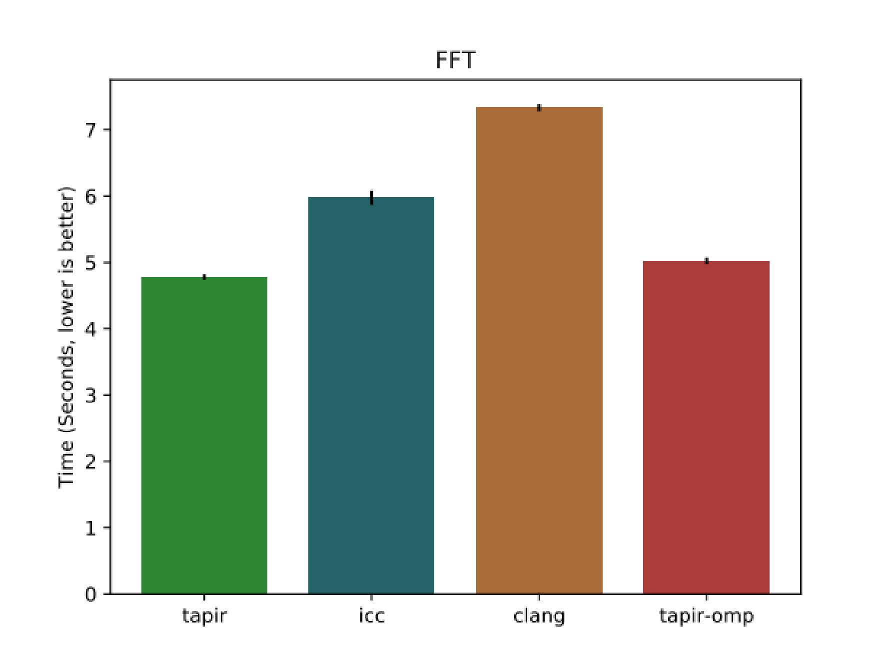
OpenMP Tasks → Tapir

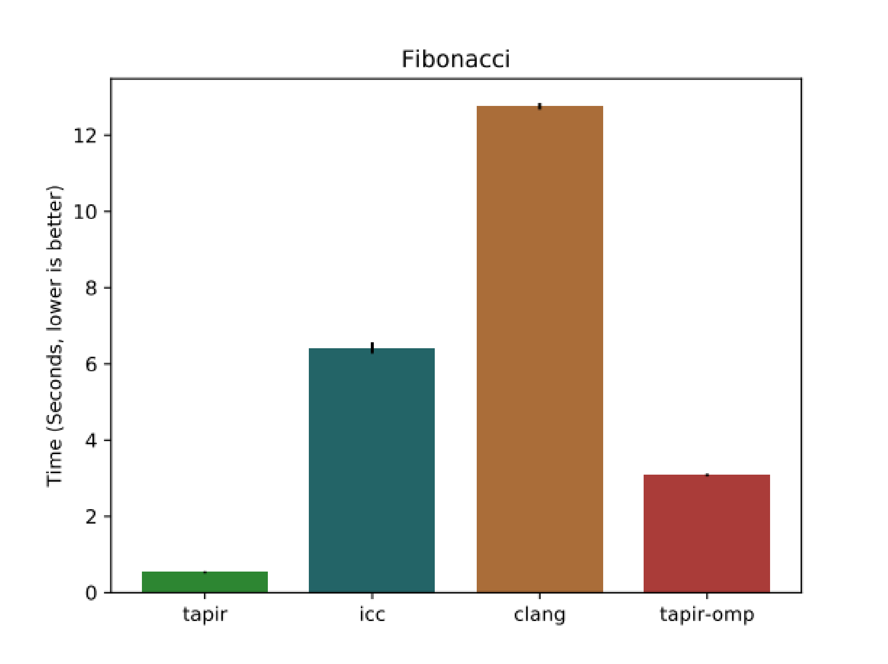
```
int fib(int n){

...
    #pragma omp task
      x = fib(n-1);
    #pragma omp task
      y = fib(n-2);
    #pragma omp taskwait
...
}
```

```
...
if.end:
  detach label %det.achd, label %det.cont

det.achd:
  %2 = load i32, i32* %n.addr, align 4
  %sub = sub nsw i32 %2, 1
  %call = call i32 @fib(i32 %sub)
  store i32 %call, i32* %x, align 4
  reattach label %det.cont

det.cont:
  detach label %det.achd1, label %det.cont4

det.achd1:
  %3 = load i32, i32* %n.addr, align 4
  %sub2 = sub nsw i32 %3, 2
  %call3 = call i32 @fib(i32 %sub2)
  store i32 %call3, i32* %y, align 4
  reattach label %det.cont4

det.cont4:
  sync label %sync.continue
...
```

OpenMP Task Results







Acknowledgments

- The LLVM community (including our many contributing vendors)
- ALCF, ANL, and DOE
- ALCF is supported by DOE/SC under contract DE-AC02-06CH11357

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation’s exascale computing imperative.