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

Initially proposed by NVidia, ORNL and LBL

Portable code with understand of non optimal performance
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.

- RAJA ([https://github.com/LLNL/RAJA](https://github.com/LLNL/RAJA))
  ```cpp
  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](https://github.com/kokkos))
  ```cpp
  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>
<thead>
<tr>
<th>adjacent_difference</th>
<th>adjacent_find</th>
<th>all_of</th>
<th>any_of</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td>copy_if</td>
<td>copy_n</td>
<td>count</td>
</tr>
<tr>
<td>count_if</td>
<td>equal</td>
<td>exclusive_scan</td>
<td>fill</td>
</tr>
<tr>
<td>fill_n</td>
<td>find</td>
<td>find_end</td>
<td>find_first_of</td>
</tr>
<tr>
<td>find_if</td>
<td>find_if_not</td>
<td>for_each</td>
<td>for_each_n</td>
</tr>
<tr>
<td>generate</td>
<td>generate_n</td>
<td>includes</td>
<td>inclusive_scan</td>
</tr>
<tr>
<td>inner_product</td>
<td>inplace_merge</td>
<td>is_heap</td>
<td>is_heap_until</td>
</tr>
<tr>
<td>is_partitioned</td>
<td>is_sorted</td>
<td>is_sorted_until</td>
<td>lexicographical_compare</td>
</tr>
<tr>
<td>max_element</td>
<td>merge</td>
<td>min_element</td>
<td>minmax_element</td>
</tr>
<tr>
<td>mismatch</td>
<td>move</td>
<td>none_of</td>
<td>nth_element</td>
</tr>
<tr>
<td>partial_sort</td>
<td>partial_sort_copy</td>
<td>partition</td>
<td>partition_copy</td>
</tr>
<tr>
<td>reduce</td>
<td>remove</td>
<td>remove_copy</td>
<td>remove_copy_if</td>
</tr>
<tr>
<td>remove_if</td>
<td>replace</td>
<td>replace_copy</td>
<td>replace_copy_if</td>
</tr>
<tr>
<td>replace_if</td>
<td>reverse</td>
<td>reverse_copy</td>
<td>rotate</td>
</tr>
<tr>
<td>rotate_copy</td>
<td>search</td>
<td>search_n</td>
<td>set_difference</td>
</tr>
<tr>
<td>set_intersection</td>
<td>set_symmetric_difference</td>
<td>set_union</td>
<td>sort</td>
</tr>
<tr>
<td>stable_partition</td>
<td>stable_sort</td>
<td>swap_ranges</td>
<td>transform</td>
</tr>
<tr>
<td>transform_exclusive_scan</td>
<td>transform_inclusive_scan</td>
<td>transform_reduce</td>
<td>uninitialized_copy</td>
</tr>
<tr>
<td>uninitialized_copy_n</td>
<td>uninitialized_fill</td>
<td>uninitialized_fill_n</td>
<td>unique</td>
</tr>
</tbody>
</table>

[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)

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

```c
void work(int i, int * restrict In) {
    #critical
    *In = *In + i;
    #barrier
    #critical
    *In = *In + i;
}
```
Solution 2: alias information propagation

```c
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

```c
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) {
        ...
    }
```
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

```c
#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

```java
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 GOMPIImpl : ParallelRegion;
class IntelPIRImpl : ParallelRegion;
class TapirImpl : ParallelRegion;
class LLVMPIRImpl : ParallelRegion;
```
Example 1: Rodinia - hotspot3D

```c
#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, 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;
          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

```c
#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
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_denergy);
        time_step(j, nelr, old_vars, vars, sf, fluxes);
    }
}

Example 3:

```c
#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

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

```c
#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

Clang C/C++ FE/Fortran/OpenCL … …

Par/Vec/Offload Prepare phase

ScalarOpts

LLVM IR

Loop distribution

Loop fusion

Loop Unrolling

Vectorization (explicit / auto)

Community's or Vendor's Vectorizer

LLVM IR

O2 & above

Community's or Vendor's Loop Optimizers

Annotated par-loop for auto-parallelization

LoopOpts

ScalarOpts

LLVM IR

LLVM IR

Lowering and Outlining for OpenMP, Autopar, Offload

O0/O1

LLVM CG

LLVM IR

O0/O1
W-Region Implementation

class **WRN** { //base class
    BasicBlock *EntryBBBlock;
    BasicBlock *ExitBBBlock;
    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...
int fib(int n) {
    ...
    #pragma omp task
    x = fib(n-1);
    #pragma omp task
    y = fib(n-2);
    #pragma omp taskwait
    ...
}

...
OpenMP Task Results

![FFT Graph](image)

- **Time (Seconds, lower is better)**
  - tapir
  - icc
  - clang
  - tapir-emp

![NQueens Graph](image)

- **Time (Seconds, lower is better)**
  - tapir
  - icc
  - clang
  - tapir-emp

![Fibonacci Graph](image)

- **Time (Seconds, lower is better)**
  - tapir
  - icc
  - clang
  - tapir-emp
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.