LLVM Parallel Intermediate Representation: Design and Evaluation using OpenSHMEM Communications

Dounia Khaldi [1] Pierre Jouvelot [2] François Irigoin [2]
Corinne Ancourt [2] Barbara Chapman [1]

[1]Department of Computer Science, University of Houston, TX

[2]MINES ParisTech, PSL Research University, France

The Second Workshop on the LLVM Compiler Infrastructure in HPC
Austin, TX, November 15, 2015

Context and Motivation

- Cilk, UPC, X10, Habanero-Java, Chapel, OpenMP, MPI, OpenSHMEM, etc
- Parallelism handling in compilers
- Towards a High-level Parallel Intermediate Representation
  - Trade-off between expressibility and conciseness of representation
  - Simplification of transformations and semantic analyses
- Huge compiler platforms: GCC (more than 7 million lines of code), LLVM (more than 1 million lines of code)
- Previous Work - SPIRE: Sequential to Parallel Intermediate Representation Extension methodology[1]

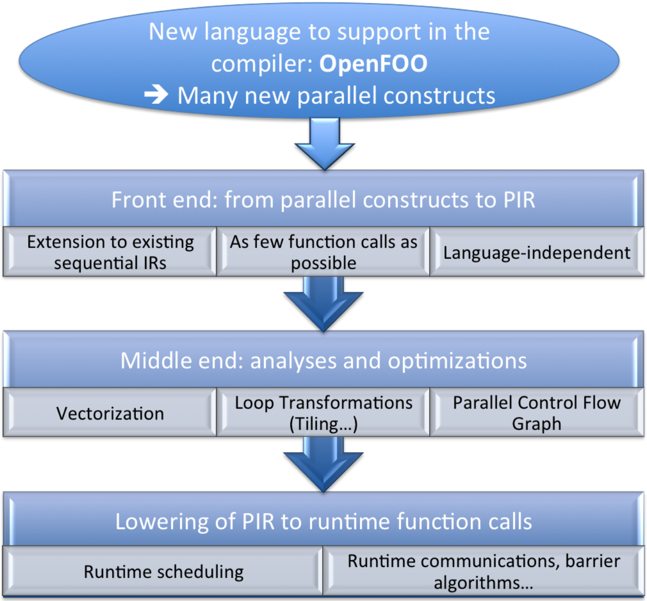
[1] Dounia Khaldi, Pierre Jouvelot, François Irigoin, Corinne Ancourt. *SPIRE: A Methodology for Sequential to Parallel Intermediate Representation Extension*. Presented at 17th Workshop on Compilers for Parallel Computing (CPC 2013), July 3-5, 2013, Lyon, France and HiPEAC Computing Systems Week, May 03, 2013, Paris, France

LLVM Compiler and its Polly Optimizer

- An open-source compilation framework that uses an intermediate representation in Static Single Assignment (SSA) form
- LLVM Optimizer (opt): Unrolling, Vectorization
- Polly, a high-level loop and data-locality polyhedral optimizer for LLVM:
  - Loop tiling, Loop fusion, Interchange, etc
  - Only Static Control Parts (SCoPs)
  - Side-effect-free functions calls
- Widely used in both academia and industry (Apple, Google, NVIDIA,...)
- Code generation for x86-64, ARM, PowerPC, Xeon Phi, etc
- Modularized pass subsystem in LLVM

LLVM logo is copyrighted by Apple Inc.

An Ideal Scenario of Parallel Compiler Design



Excerpt of LLVM IR Grammar

```
function     = blocks:block*;
block        = label:identifier x phi_nodes:phi* x
               instructions:instruction* x terminator;
phi          = call;
instruction  = load + store + call;
load         = identifier;
store        = name:identifier + value:expression;
terminator   = cond_br + uncond_br + return;
cond_br      = value:identifier x label_true:identifier x
               label_false:identifier;
uncond_br    = label:identifier;
return       = value:identifier;
```

```
sum = 42;
for(i=0;i<10;i++){
  sum = sum + 2;
}
```

```
entry:
  ...
 br label %bb1
bb:             ; preds = %bb1
 %0 = add nsw i32 %sum.0, 2
 %1 = add nsw i32 %i.0, 1
 br label %bb1
bb1:            ; preds = %bb, %entry
 %sum.0=phi i32 [42,%entry],[%0,%bb]
 %i.0 = phi i32 [0,%entry],[%1,%bb]
 %2 = icmp sle i32 %i.0, 10
 br i1 %2, label %bb, label %bb2
bb2:       ; preds = %bb1
```

LLVM PIR: Execution

- Add execution information to control constructs:
  - `function`
  - `block`

```
execution  = sequential:unit + parallel:scheduling +
             reduced:unit;

scheduling = static:unit + dynamic:unit +
             speculative:unit + default:unit
```

- `sections` in OpenMP, and its LLVM PIR representation (simplified):

```
#pragma omp sections nowait
{
  #pragma omp section
  x = foo();
  #pragma omp section
  y = bar();
}
z = baz(x,y);
```

```
block(x = foo();
      y = bar(),
      parallel);
block(z = baz(x,y),
      sequential)
```

LLVM PIR: Collective Synchronization

- Add `synchronization` information to `block` and `instruction` domains:

```
synchronization = none:unit +
                  spawn:identifier + barrier:unit +
                  atomic:identifier;
```

- Cilk and OpenMP variants of atomic synchronization on the reference `l`

```
Cilk_lockvar l;
Cilk_lock_init(l);
...
Cilk_lock(l);
 x[index[i]] += f(i);
Cilk_unlock(l);
```

```
#pragma omp critical
  x[index[i]] += f(i);
```

LLVM PIR: Point-to-Point Synchronization (Event API)

- Add event as a new type:

```
event newEvent(int i);
void signal(event e);
void wait(event e);
```

- A phaser in Habanero-Java, and its LLVM PIR representation (simplified):

```
finish{
 phaser ph=new phaser();
 for(j = 1;j <= n;j++){
     async phased(
         ph<SIG_WAIT>){
            S; next; S′;
     }
 }
}
```

```
barrier(
  ph=newEvent(-(n-1));
  forloop(j, 1, n, 1,
          spawn(j, S;
                signal(ph);
                wait(ph);
                signal(ph);
                S′),
          parallel);
)
```

LLVM PIR: One-Sided Communications

- Add `location` information to `identifier` domain:

```
location = private:unit + shared:unit +
           pgas:unit;
```

- put in OpenSHMEM, and its LLVM PIR representation (simplified):

`shmem_int_put (dest, src, 20, pe);`	`for(i=0; i<20; i++) dest{pe}[i]= src[i];`
`shmem_int_get (dest, src, 20, pe);`	`for(i=0; i<20; i++) dest[i]= src{pe}[i];`

where `location` of `dest/src` is pgas

- `identifier` domain extended to handle expressions in the left hand side of assignments

LLVM PIR: Summary

```
function      = blocks:block*;
block         = label:identifier x phi_nodes:phi* x
                instructions:instruction* x terminator;
phi           = call;
instruction   = load + store + call;
load          = identifier;
store         = name:identifier + value:expression;
terminator    = cond_br + uncond_br + return;
cond_br       = value:identifier x label_true:identifier x
                label_false:identifier;
uncond_br     = label:identifier;
return        = value:identifier;
```

```
function    ~> function x execution;
block       ~> block x execution x synchronization;
instruction ~> instruction x synchronization;
load        ~> load x expression;
store       ~> store x expression;
identifier  ~> identifier x location;
type        ~> type + event:unit

Intrinsic Functions: send, recv, signal, wait
```

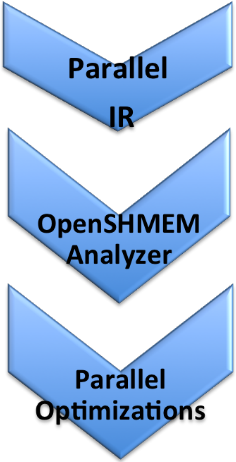
Overview of OpenSHMEM

- OpenSHMEM: Light-weight and portable PGAS Library
- SPMD-like style of programming
- Properties available in recent OpenSHMEM-1.2 specifications:
  - Symmetric Data Object management
  - Remote Read and Write using Put and Get operations
  - Barrier and Point-to-Point synchronization
  - Atomic memory operations and collectives

```
int src;
int *dest;
...
shmem_init();
...
src  = me;
dest = (int *) shmem_malloc(sizeof (*dest));
nextpe = (me + 1) % npes;  /*wrap around */

shmem_int_put (dest, &src, 1, nextpe);
more_work_goes_here (...);
shmem_barrier_all();
x = dest * 0.995 + 45 * y;
...
```

Towards OpenSHMEM (PGAS) Analyzer and Optimizer



- LLVM IR, WHIRL
- PGAS programming models

- Semantic awareness of OpenSHMEM in the compiler
- Comprehensive analysis and optimization framework for OpenSHMEM

- Communication vectorization
- Communication/computation overlap

- SPIRE(LLVM IR) yields LLVM PIR
- shmem put/get ⇝ load/store operations
- Remote PEs ⇝ LLVM Metadata
- To avoid dead code elimination ⇝ volatile load/store
- Example: LLVM PIR of `shmem_int_put(dest, src, N, pe)`

```
LoadStoreLoop:                  ; preds = %LoadStoreLoop, %entry
  %lafee = phi i64 [0, %entry], [%nextvar, %LoadStoreLoop]
  %addressSrc = getelementptr i32* getelementptr
         inbounds([11 x i32]* @src,i32 0,i32 0),i64 %lafee
  %addressDst = getelementptr i32* getelementptr
         inbounds([11 x i32]* @dest,i32 0,i32 0),i64 %lafee
  %RMA = load i32* %addressSrc
  store volatile i32 %RMA, i32* %addressDst, !PE !{i32 %7}
  %nextvar = add i64 %lafee, 1
  %cmptmp = icmp ult i64 %nextvar, %N
  br i1 %cmptmp, label %LoadStoreLoop, label %shmem_RDMA_bb
```

Example of Loop Unrolling from IS (NPBs)

- Loops have to be canonicalized: use Polly
- Polly: No built-ins $\leadsto$ more SCoPs

```
for(j=0;j<num_pes;j++)
  shmem_int_put(&recv_count[my_rank],&send_count[j],1,j);
```

```
for(j=0;j<num_pes;j++)
  recv_count[my_rank]{j} = send_count[j]
```

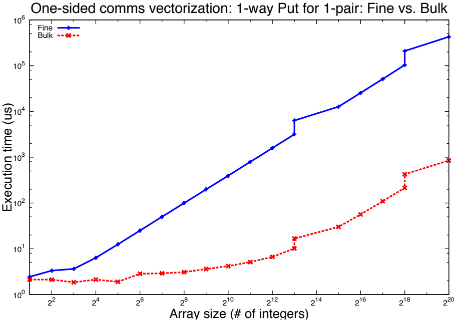
- `-loop-unroll` on `shmem_int_put` $\Rightarrow$ No unrolling
- `-loop-unroll` on `load/store` $\Rightarrow$ Unrolling of 8

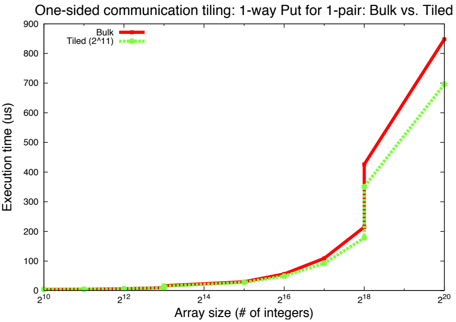
```
opt -load ${LLVM_BUILD}/lib/LLVMPolly.so -S -polly-canonicalize
          is.s -basicaa -loop-unroll  -o is.unroll.bc
```

```
for(j=0;j<num_pes;j+=8){
  shmem_int_put(&recv_count[my_rank],&send_count[j],1,j);
  shmem_int_put(&recv_count[my_rank],&send_count[j+1],1,j+1);
  shmem_int_put(&recv_count[my_rank],&send_count[j+2],1,j+2);
  shmem_int_put(&recv_count[my_rank],&send_count[j+3],1,j+3);
  shmem_int_put(&recv_count[my_rank],&send_count[j+4],1,j+4);
  shmem_int_put(&recv_count[my_rank],&send_count[j+5],1,j+5);
  shmem_int_put(&recv_count[my_rank],&send_count[j+6],1,j+6);
  shmem_int_put(&recv_count[my_rank],&send_count[j+7],1,j+7);
}
```

Optimizations of RMAs using Microbenchmarks

- Testing optimizations on OpenSHMEM in LLVM
- One-sided Communication Optimizations:
  - Communication vectorization (Modified LoopIdiomRecognize pass)
  - Communication strip mining (Strip size = 2048)
  - Unrolling
- Stampede SuperComputer
- MVAPICH2-X, version 2.0b
- LLVM-3.5.0, sporting the same version of Polly





Conclusion

- *"Parallelism or concurrency are operational concepts that refer not to the program, but to its execution." [Dijkstra, 1977]*
- SPIRE(LLVM IR) $\rightsquigarrow$ LLVM PIR
- Trade-off between expressibility and conciseness of representation
- Generality to represent the constructs of current parallel languages
- Implementation of RMA PIR in LLVM and application using OpenSHMEM
- Polly: No built-ins $\rightsquigarrow$ more SCoPs
- Communication vectorization, Strip mining, Unrolling

Future Work

- Applying other LLVM existing optimizations on OpenSHMEM programs
- Computations/communications overlapping, parallel side effects, etc

LLVM Parallel Intermediate Representation: Design and Evaluation using OpenSHMEM Communications

Dounia Khaldi [1] Pierre Jouvelot [2] François Irigoin [2]
Corinne Ancourt [2] Barbara Chapman [1]

[1]Department of Computer Science, University of Houston, TX

[2]MINES ParisTech, PSL Research University, France

The Second Workshop on the LLVM Compiler Infrastructure in HPC
Austin, TX, November 15, 2015