Molly: Parallelizing for Distributed Memory using LLVM

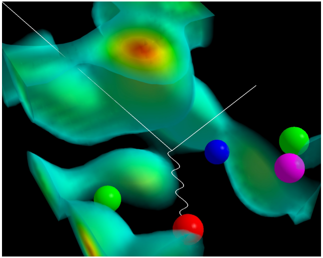
Michael Kruse

INRIA Saclay Île-de-France/Laboratoire de Recherche en Informatique/Ecole Normale Superieur

17 March 2016

Supported by: Université Paris-Sud XI, INRIA, project MANY

Quantum Chromodynamics (QCD)



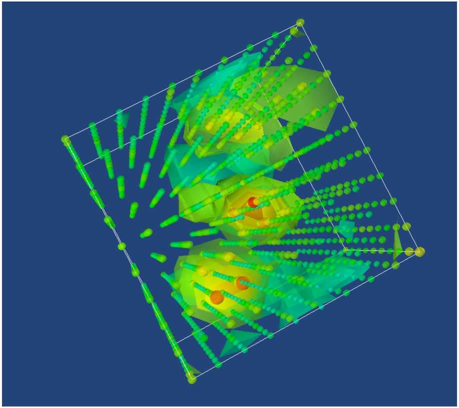
(src: Derek Leineweber)

$$F_{\mu\nu}(x) = \partial_\mu A_\nu(x) - \partial_\nu A_\mu(x) + ig[A_\mu(x), A_\nu(x)]$$

$$\mathcal{L} = \underbrace{\sum_q \bar{\psi}_q[\gamma_\mu(\partial_\mu - igA_\mu) + m_q]\psi_q}_{\text{fermionic action}} + \underbrace{\frac{1}{4}F_{\mu\nu}^2}_{\text{gauge action}}$$

- Quantum Field Theory (QFT)
- Strong force
- Quarks (up, down, strange, charm, top, bottom)
- Mediator particle: Gluons (8 colors)

Lattice QCD



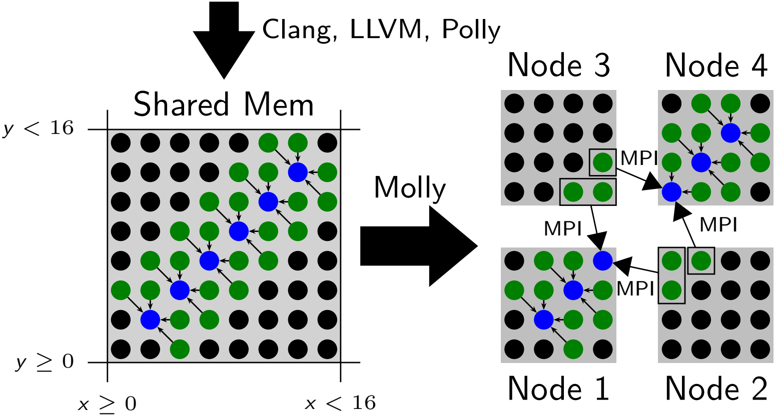
(src: Richard C. Brower, SciDAC)

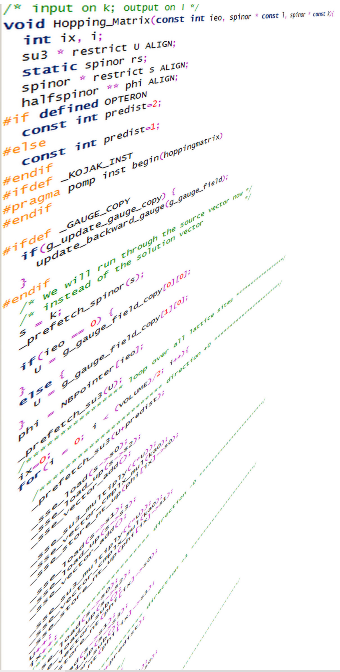
- Discretization of Space-Time (4-dimensional)
- Larger the lattice $\rightarrow$ smaller distances $\rightarrow$ more accurate results

Molly: Semi-Automatic Memory Distribution with LLVM

```
#pragma molly transform("{ [x,y] -> [node[x/4,y/4], local[x%4,y%4]] }")
molly::array<double,16,16> field;

for (int i = 1; i < 15; i += 1)
   field[i][i] = field[i+1][i] + field[i+1][i-1] + field[i][i+1] + field[i-1][i+1];
```




/* input on k; output on l */
void Hopping_Matrix(const int ieo, spinor * const l, spinor * const k){
int ix, i;
su3 * restrict U ALIGN;
static spinor rs;
spinor * restrict s ALIGN;
halfspinor ** phi ALIGN;
#if defined OPTERON
const int predist=2;
#else
const int predist=1;
#endif
#ifdef _KOJAK_INST
#pragma pomp inst begin(hoppingmatrix)
#endif
#ifdef _GAUGE_COPY
if(g_update_gauge_copy) {
update_backward_gauge(g_gauge_field);
}
#endif

Static Control Part (SCoP)

```
    for (int i = 0; i < 6; ++i)
      for (int j = 0; j < 6; ++j)
S1:     A[i][j] = 0;
    for (int i = 0; i < 5; ++i)
      for (int j = 0; j < 5; ++j)
S2:     B[i][j] = A[i+1][j] + A[i][j+1];
```

- $\Omega = \{\mathsf{S1}, \mathsf{S2}\}$
- $\mathcal{D} = \{(\mathsf{S1}, (i, j)) \mid 0 \leq i, j < 6\} \cup \{(\mathsf{S2}, (i, j)) \mid 0 \leq i, j < 5\}$

Static Control Part (SCoP)

```
    for (int i = 0; i < 6; ++i)
      for (int j = 0; j < 6; ++j)
S1:     A[i][j] = 0;
    for (int i = 0; i < 5; ++i)
      for (int j = 0; j < 5; ++j)
S2:     B[i][j] = A[i+1][j] + A[i][j+1];
```

- $\mathcal{V} = \{\mathtt{A}, \mathtt{B}\}$
- $\mathcal{E} = \{(\mathtt{A}, (x, y)) \mid 0 \leq x, y < 6\} \cup \{(\mathtt{B}, (x, y)) \mid 0 \leq x, y < 6\}$

Static Control Part (SCoP)

```
    for (int i = 0; i < 6; ++i)
      for (int j = 0; j < 6; ++j)
S1:     A[i][j] = 0;
    for (int i = 0; i < 5; ++i)
      for (int j = 0; j < 5; ++j)
S2:     B[i][j] = A[i+1][j] + A[i][j+1];
```

- $<_{\text{flow}} = \{(\text{S1}, (i_1, j_2)) \mapsto (\text{S2}, (i_2, j_2)) \mid (i_1 = i_2 + 1, j_1 = j_2) \vee (i_1 = i_2, j_1 = j_2 + 1)\} \cup \{\ldots$
  - **G**enerator $\rightarrow$ **C**onsumer
- $<_{\text{dep}} = <_{\text{flow}} \cup <_{\text{anti}} \cup <_{\text{output}}$

Extensions

Prologue and Epilogue

```
    ⊤
    for (int i = 0; i < 6; ++i)
      for (int j = 0; j < 6; ++j)
S1:     A[i][j] = 0;
    for (int i = 0; i < 5; ++i)
      for (int j = 0; j < 5; ++j)
S2:     B[i][j] = A[i+1][j] + A[i][j+1];
    ⊥
```

- $\Omega' = \Omega \cup \{\top, \bot\}$

Extensions

Prologue and Epilogue

```
    ⊤
    for (int i = 0; i < 6; ++i)
      for (int j = 0; j < 6; ++j)
S1:     A[i][j] = 0;
    for (int i = 0; i < 5; ++i)
      for (int j = 0; j < 5; ++j)
S2:     B[i][j] = A[i+1][j] + A[i][j+1];
    ⊥
```

- $\Omega' = \Omega \cup \{\top, \bot\}$
- $\lambda_{\mathsf{write},\top} = \mathcal{E}$
- $\lambda_{\mathsf{read},\bot} = \mathcal{E}$

Extensions

Variables and Fields

```
    for (int i = 0; i < 6; ++i)
      for (int j = 0; j < 6; ++j)
S1:      A[i][j] = 0;
    for (int i = 0; i < 5; ++i)
      for (int j = 0; j < 5; ++j)
S2.1:    tmp1 = A[i+1][j];
S2.2:    tmp2 = A[i][j+1];
S2.3:    B[i][j] = tmp1 + tmp2;
```

- $\mathcal{F} = \{\mathtt{A}, \mathtt{B}\}$
- $\mathcal{V} = \mathcal{F} \cup \{\mathtt{tmp1}, \mathtt{tmp2}\}$

Data Distribution

Array Element's Home Location

- On which node to store a value?
  (While not executing a SCoP;
  in a SCoP it's the node that computed the value)

Data Distribution

Array Element's Home Location

- On which node to store a value?
  (While not executing a SCoP;
  in a SCoP it's the node that computed the value)
- Alignment problem
- Not focus of this work
- Default policy: Block-distribution

Work Distribution

Processor Executing Code

- On which node to execute a statement instance?
  (Non-SCoP code executed as SPMD)

Work Distribution

Processor Executing Code

- On which node to execute a statement instance?
  (Non-SCoP code executed as SPMD)
- Related to alignment
- Not focus of this work
- Default policy:
  1. Definitions of non-field elements are executed everywhere
  2. Owner computes
  3. Dependence computes
  4. Master computes (last resort)

Chunking

- Which values to put into the same message?

Chunking

- Which values to put into the same message?

Definition

Chunking Function A function $\varphi : <_{\text{flow}} \rightarrow \mathcal{X}$; two values of a flow belong to the same message iff the chunking function maps to the same value (and source and target node are the same)

- Independent of actual distribution
- The chunking space $\mathcal{X}$ can be chosen arbitrarily
- Trivial chunking function: $\varphi((G, \vec{i}), (C, \vec{j})) = (G, \vec{i}, C, \vec{j})$
- Put value just once per message: $\varphi((G, \vec{i}), (C, \vec{j})) = (G, \vec{i})$

What to Optimize for?

- Largest messages
- Fewest number of transfers
- Balanced message sizes
- Longest time between send and wait
- Shortest transmission distance
- Lowest runtime overhead
- Programmer control
- ...

What to Optimize for?

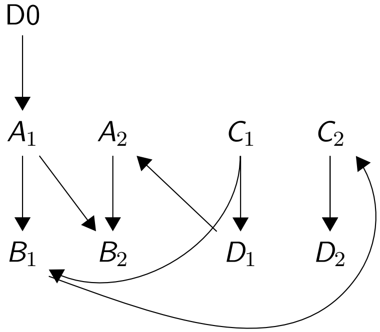
- Largest messages
- Fewest number of transfers
- Balanced message sizes
- Longest time between send and wait
- Shortest transmission distance
- Lowest runtime overhead
- Programmer control
- ...

1) Antichain Chunking

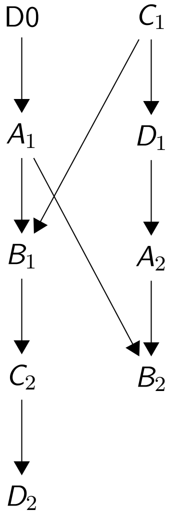
1. Create the detailed dependence graph

   (completely unrolled, graph node = statement instance)
2. Align around longest path/chain
3. Chunk antichains together
4. Minimum antichain decomposition = Longest chain (**Mirsky's Theorem**)

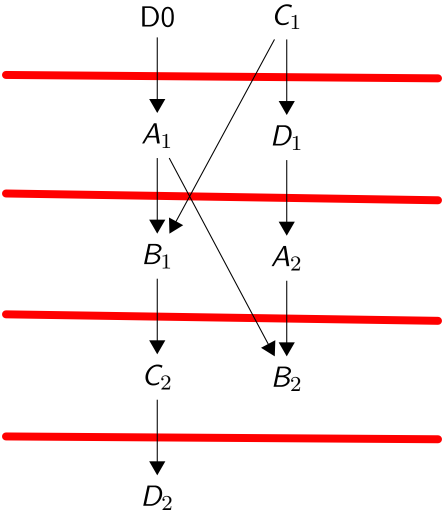
Example: Antichain Chunking



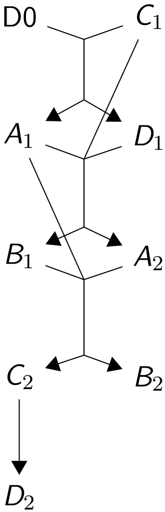
Example: Antichain Chunking



Example: Antichain Chunking



Example: Antichain Chunking



2) Parametric Chunking

- Antichain Chunking is restricted to non-parametric dependence graphs
- Use IntLP algorithms for parametric version
- Difference to normal scheduling: Non-flow dependencies do not impose sequentialization

- Normal dependence condition for $u <_{\mathsf{dep}} v$:
  $\theta(u) - \theta(v) \ll_{\mathsf{lex}} \vec{0}$

2) Parametric Chunking

- Antichain Chunking is restricted to non-parametric dependence graphs
- Use IntLP algorithms for parametric version
- Difference to normal scheduling: Non-flow dependencies do not impose sequentialization

- Dependence condition for $u <_{\text{flow}} v$:
  $\theta(u) - \theta(v) \ll_{\text{lex}} \vec{0}$
- Dependence condition for $u\ (<_{\text{dep}} \setminus <_{\text{flow}})\ v$:
  $\theta(u) - \theta(v) \leqslant_{\text{lex}} \vec{0}$
- If already known, node-local flows can be removed from $<_{\text{flow}}$
- Run scheduling algorithm (Feautrier's Farkas Algorithm, etc.)
  - Result is our chunking relation $\varphi : \mathcal{D} \to \mathbb{Z}^n$
  - $\mathcal{X} := \mathbb{Z}^n$ is our chunking space

Example: Schedule-Dependent Chunking

```
  for (int k = 0; k < 6; ++k)
    for (int i = 0; i < 6; ++i)
S1:     A[i] = f(k,i);
    for (int i = 0; i < 6; ++i)
      for (int j = 0; j < 6; ++j)
S2:       B[i][j] = A[i] + A[j];
```

Example: Schedule-Dependent Chunking

```
for (int k = 0; k < 6; ++k)
    for (int i = 0; i < 6; ++i)
S1:    A[i] = f(k,i);
    for (int i = 0; i < 6; ++i)
      for (int j = 0; j < 6; ++j)
S2:     B[i][j] = A[i] + A[j];
```

$\varphi_{\mathrm{S1}}(k,i) = (\mathrm{S1},k); \quad \varphi_{\mathrm{S2}}(k,i,j) = (\mathrm{S2},k);$

Transfer Primitives on Buffers

- **send_wait**: Wait until we can overwrite data in a buffer
  - Execute instances that write to buffer
- **send**: Send message in buffer
- **recv_wait**: Wait until message has arrived
  - Execute instances that read from buffer
- **recv**: Reset buffer to accept next recurring message

Transfer Primitives on Buffers

- **get_input**: Load required data from home memory (**prologue**)
- **send_wait**: Wait until we can overwrite data in a buffer
  - Execute instances that write to buffer
- **send**: Send message in buffer
- **recv_wait**: Wait until message has arrived
  - Execute instances that read from buffer
- **recv**: Reset buffer to accept next recurring message
- **writeback**: Write data to persistent home memory (**epilogue**)

Modified Program

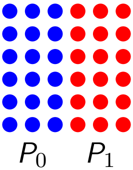
```
    send_wait(combufB);
    for (int k = 0; k < 6; ++k)
      sendwait(combufA, k);
      for (int i = 0; i < 6; ++i)
S1:     combufA[i] = f(k,i);
      send(combufA, k);
      recvwait(combufA, k);
      for (int i = 0; i < 6; ++i)
        for (int j = 0; j < 6; ++j)
S2:         combufB[i][j] = combufA[i] + combufA[j];
      recv(combufA, k);
    recvwait(combufB);
    send(combufB);
    recvwait(combufB);
    for (int i = 0; i < 6; ++i)
      for (int j = 0; j < 6; ++j)
        B[i][j] = combufB[i][j];
    recv(combufB)
```

Epilogue

Add Distribution

Block Distribution

- $\mathcal{P} = \{0, 1\}$
- Element distribution
  - $\pi_A = \{((x, y), p) \mid 3p \leq x < 3(p + 1), 0 \leq y < 6\}$
  - $\pi_B = \{((x, y), p) \mid 3p \leq x < 3(p + 1), 0 \leq y < 6\}$
  - `A[0..2,0..5]` $\rightarrow P_0$, `A[3..5,0..5]` $\rightarrow P_1$
- Instance distribution
  - $\pi_{S1} = \{(i, p) \mid 3p \leq i < 3(p + 1)\}$
  - $\pi_{S2} = \{((i, j), p) \mid 3p \leq i < 3(p + 1), 0 \leq j < 6\}$



Modified Program with Distribution

P_0:

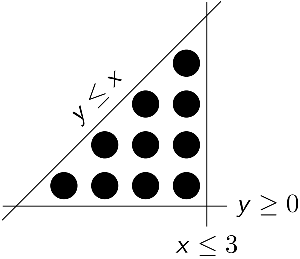
```
    for (int k = 0; k < 6; ++k)
      send_wait(combufA, k, P1)
      for (int i = 0; i < 3; ++i)
S1.1:   tmp1 = f(k,i);
S1.2:   A[i] = tmp1;
S1.2:   combufA[i] = tmp1;
      send(combufA, k, P1);
      for (int i = 0; i < 3; ++i)
        for (int j = 0; j < 6; ++j)
S2.1:     tmp2 = A[i];
          if (0 <= j < 3)
S2.2:       tmp3 = A[j];
          if (i == 0 && j == 3)
            recv_wait(combufA, k, P1);
          if (3 <= j < 6)
S2.2:       tmp3 = combufA[j];
          if (i == 3 && j == 5)
            recv(combufA, k, P1);
S2.3:     B[i][j] = tmp2 + tmp3;
```

P_1:

```
    for (int k = 0; k < 6; ++k)
      send_wait(combufA, k, P0)
      for (int i = 3; i < 6; ++i)
S1.1:   tmp1 = f(k,i);
S1.2:   A[i] = tmp1;
S1.2:   combufA[i] = tmp1;
      send(combufA, k, P0);
      for (int i = 3; i < 6; ++i)
        for (int j = 0; j < 6; ++j)
S2.1:     tmp2 = A[i];
          if (3 <= j < 6)
S2.2:       tmp3 = A[j];
          if (i == 3 && j == 0)
            recv_wait(combufA, k, P0);
          if (0 <= j < 3)
S2.2:       tmp3 = combufA[j];
          if (i == 5 && j == 5)
            recv(combufA, k, P0);
S2.3:     B[i][j] = tmp2 + tmp3;
```

Message Size and Mapping

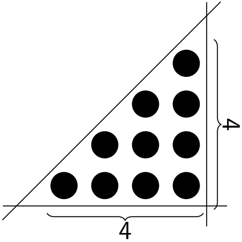
Exact Method



- Using Ehrhart (Quasi-)Polynomials
- Size: Total number of elements
  - $\sum_{x=0}^{3} \sum_{y=0}^{x} 1 = 10$
- Index of (x, y) = Number of coordinates that are lexicographically smaller than (x, y)
  - $\sum_{x'=0}^{x-1} (x' + 1) + y = \frac{(x+1)x}{2} + y$
- Barvinok library

Message Size and Mapping

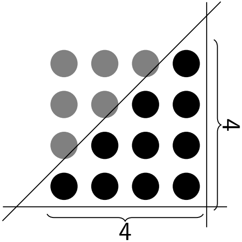
Bounding Box Method



- Using bounding box of contained elements ($\rightarrow$ rectangular array)
  - Wasted space due to unused elements
  - Simpler index calculation
- Size: $4 \times 4 = 16$ (6 unused)
- Index: $4x + y$ (row major)

Message Size and Mapping

Bounding Box Method



- Using bounding box of contained elements ($\rightarrow$ rectangular array)
  - Wasted space due to unused elements
  - Simpler index calculation
- Size: $4 \times 4 = 16$ (6 unused)
- Index: $4x + y$ (row major)

Molly

- Clang
  - Parse
  - IR Generation
- LLVM
  - IR handling infrastructure
  - Backend
- Integer Set Library (ISL)
  - Vector sets represented by polyhedra
- Polly
  - SCoP analysis
  - Optimization
  - IR Re-Generation
- Molly
  - `molly.h`
  - *Transfer Code* generator
  - MollyRT (MPI/SPI Backend)

C++ Language Extensions

1. ```
   #include <molly.h>
   molly::array<double,6,6> A;
   ```
   - Namespace `molly`
   - Variadic template `array`
   - Recursively overloaded `operator[]`
   - Array elements as separate allocations
   - Don't store the element pointer
   ```

C++ Language Extensions

1. ```
   #include <molly.h>
   molly::array<double,6,6> A;
   ```
   - Namespace `molly`
   - Variadic template `array`
   - Recursively overloaded `operator[]`
   - Array elements as separate allocations
   - Don't store the element pointer

2. ```
   #pragma molly transform("{[x,y] -> [node[floor(x/3),floor(y/3)], local[x,y]]}")
   ```
   - Set home location for each array element
   - Multiple home locations possible

C++ Language Extensions

1. `#include <molly.h>`
   `molly::array<double,6,6> A;`
   - Namespace `molly`
   - Variadic template `array`
   - Recursively overloaded `operator[]`
   - Array elements as separate allocations
   - Don't store the element pointer

2. `#pragma molly transform("{[x,y] -> [node[floor(x/3),floor(y/3)], local[x,y]]}")`
   - Set home location for each array element
   - Multiple home locations possible

3. `#pragma molly where("{[i,j] -> node[floor(i/3),floor(j/3)]}")`
   - Set execution location of following statement

Input Example

Command

```
$ mollycc flow.cpp -mllvm -shape=2x1 -emit-llvm -S
```

```cpp
#include <molly.h>

#pragma molly transform("{ [x,y] -> [node[x/4], local[floord(x%4)]] }")
molly::array<double,8,8> A;

double [[molly::pure]] f(int,int);
void [[molly::pure]] g(int,int,double);

void flow() {
  for (int i = 0; i < 8; i += 1)
    for (int j = 0; j < 8; j += 1)
      A[i][j] = f(i,j);

  for (int i = 0; i < 8; i += 1)
    for (int j = 0; j < 8; j += 1)
#pragma molly where("{ [x,y] -> node[y/4] }")
      g(i,j,A[i][j]);
}
```

Clang Output

```
@A = global %"class.molly::array" zeroinitializer, align 8

define internal void @_GLOBAL__sub_I_flow.cpp() {
entry:
  call void @"\01??__Eignore@std@@YAXXZ"()
  call void @"\01??__Edata@@YAXXZ"()
  ret void
}

define internal void @"\01??__Edata@@YAXXZ"() {
entry:
  call void @llvm.molly.field.init(%"class.molly::array"* @A, metadata !10)
  ret void
}

define void @flow() #6 {
entry:
  br label %for.cond

for.body3:
  %stval = call double @f(i32 %i, i32 %j)
  %stptr = call double* @llvm.molly.ptr(%"class.molly::array"* %A, %i, %j)
  store double %stval, double* %stptr
  br label %for.inc

where.body:
  %ldptr = call double* @llvm.molly.ptr(%"class.molly::array"* %A, %i, %j), !where !11
  %ldval = load double* %ldptr, !where !11
  call void @g(i32 %i, i32 %j, %ldval), !where !11
  br label %where.end

for.end25:
  ret void
}

@llvm.global_ctors = appending global [1 x { i32, void ()*, i8* }] [{ i32, void ()*, i8* } { i32 65535, void ()* @_GLOBAL__sub_I_flow.cpp, i8* null }]

!molly.fields = !{!7}
!molly.runtime = !{!9}

!7 = metadata !{metadata !"field", metadata !"array", metadata !"class.molly::array", i64 12, metadata !8, null, null, void (%"class.molly::array"*, double*
!8 = metadata !{i32 8, i32 4}
!9 = metadata !{%"class.molly::SendCommunicationBuffer"* null, %"class.molly::RecvCommunicationBuffer"* null, i32* null, void (%"class.molly::SendCommunicat
!10 = metadata !{metadata !"fieldvar", metadata !"{ [x,y] -> [node[x/4], local[floord(x%4)]] }", i64 0}

!11 = metadata !{metadata !"where", metadata !"{ [x,y] -> node[y/4] }"}
```

Clang Output

MollyRT

```
!molly.runtime = !{!9}

!9 = metadata !{%"class.molly::SendCommunicationBuffer"* null,
  %"class.molly::RecvCommunicationBuffer"* null, i32* null,
  void (%"class.molly::SendCommunicationBuffer"*, i32, i32)* @__molly_sendcombuf_create,
  void (%"class.molly::RecvCommunicationBuffer"*, i32, i32)* @__molly_recvcombuf_create,
  null, null, i32 (i32)* @__molly_local_coord}
```

- Types use by runtime (eg. what's representing a node's *rank*)
- Allocation functions to call

Clang Output

Fields

Source

```
#pragma molly transform("{ [x,y] -> [node[x/4], local[floord(x%4)]] }")
molly::array<double,8,8> A;
```

```
@A = global %"class.molly::array" zeroinitializer, align 8
```

Clang Output

Fields

Source

```
#pragma molly transform("{ [x,y] -> [node[x/4], local[floord(x%4)]] }")
molly::array<double,8,8> A;
```

```
@A = global %"class.molly::array" zeroinitializer, align 8

@llvm.global_ctors = appending global [1 x { i32, void ()*, i8* }]
  [{ i32, void ()*, i8* } { i32 65535, void ()* @_GLOBAL__sub_I_flow.cpp, i8* null }]

define internal void @_GLOBAL__sub_I_flow.cpp() {
entry:
  call void @"\01??__Eignore@std@@YAXXZ"()
  call void @"\01??__Edata@@YAXXZ"()
  ret void
}

define internal void @"\01??__Edata@@YAXXZ"() {
entry:
  call void @llvm.molly.field.init(%"class.molly::array"* @A, metadata !10)
  ret void
}

!10 = metadata !{metadata !"fieldvar", metadata
  !"{ [x,y] -> [node[floor(x/4),floor(y/4)] -> local[x,y]] }", i64 0}
```

Clang Output

Fields

Source

```
#pragma molly transform("{ [x,y] -> [node[x/4], local[floord(x%4)]] }")
molly::array<double,8,8> A;
```

```
@A = global %"class.molly::array" zeroinitializer, align 8

!molly.fields = !{!7}

!7 = metadata !{metadata !"field", metadata !"array", metadata !"class.molly::array",
  i64 12, metadata !8, null, null,
  void (%"class.molly::array"*, double*, i64, i64)* @"\01?__get_broadcast@?$array@N$07$03@molly@@QEBAXAEAN_
  void (%"class.molly::array"*, double*, i64, i64)* @"\01?__set_broadcast@?$array@N$07$03@molly@@QEAAXAEBN_
  void (%"class.molly::array"*, double*, i64, i64)* @"\01?__get_master@?$array@N$07$03@molly@@QEBAXAEAN_J1@
  void (%"class.molly::array"*, double*, i64, i64)* @"\01?__set_master@?$array@N$07$03@molly@@QEBAXAEBN_J1@
  i1 (%"class.molly::array"*, i64, i64)* @"\01?isLocal@?$array@N$07$03@molly@@QEBA_N_J0@Z",
  double* null, i64 8,
  double* (%"class.molly::array"*, i64, i64)* @"\01?__ptr_local@?$array@N$07$03@molly@@QEAAPEAN_J0@Z"}
!8 = metadata !{i32 8, i32 4}
!10 = metadata !{metadata !"fieldvar", metadata
  !"{ [x,y] -> [node[floor(x/4),floor(y/4)] -> local[x,y]] }", i64 0}
```

- Array type
- Array dimensions/size-per-dimension
- Element type/size
- Methods expected to be implemented by compiler

Clang Output

Array access

Source

```
for (int i = 0; i < 8; i += 1)
  for (int j = 0; j < 8; j += 1)
    A[i][j] = f(i,j);
for (int i = 0; i < 8; i += 1)
  for (int j = 0; j < 8; j += 1)
    g(i,j,A[i][j]);
```

```
for.body3:
  %stval = call double @f(i32 %i, i32 %j)
  %stptr = call double* @llvm.molly.ptr(%"class.molly::array"* %A, %i, %j)
  store double %stval, double* %stptr
  br label %for.inc

where.body:
  %ldptr = call double* @llvm.molly.ptr(%"class.molly::array"* %A, %i, %j)
  %ldval = load double* %ldptr
  call void @g(i32 %i, i32 %j, %ldval)
  br label %where.end
```

Clang Output

Where clause

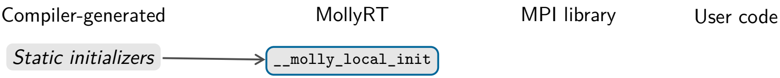
Source

```
#pragma molly where("{ [x,y] -> node[y/4] }")
  g(i,j,A[i][j]);
```

```
where.body:
  %ldptr = call double* @llvm.molly.ptr(%"class.molly::array"* %A, %i, %j), !where !11
  %ldval = load double* %ldptr, !where !11
  call void @g(i32 %i, i32 %j, %ldval), !where !11
  br label %where.end

!11 = metadata !{metadata !"where", metadata !"{ [x,y] -> node[y/4] }"}
```

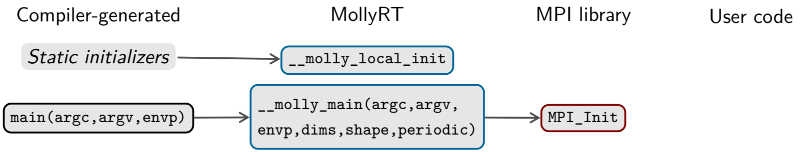
MollyRT

Init- and finalization



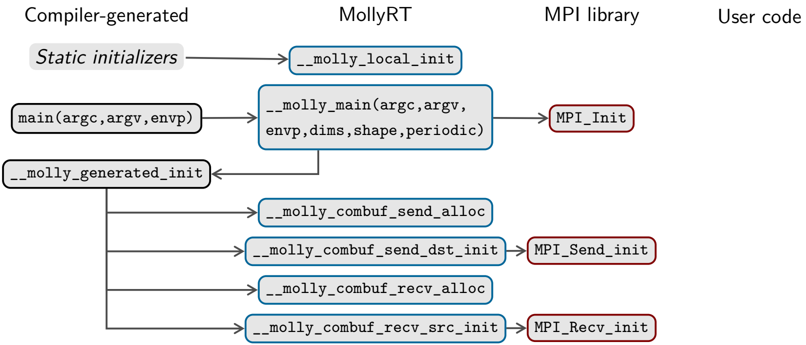
MollyRT

Init- and finalization



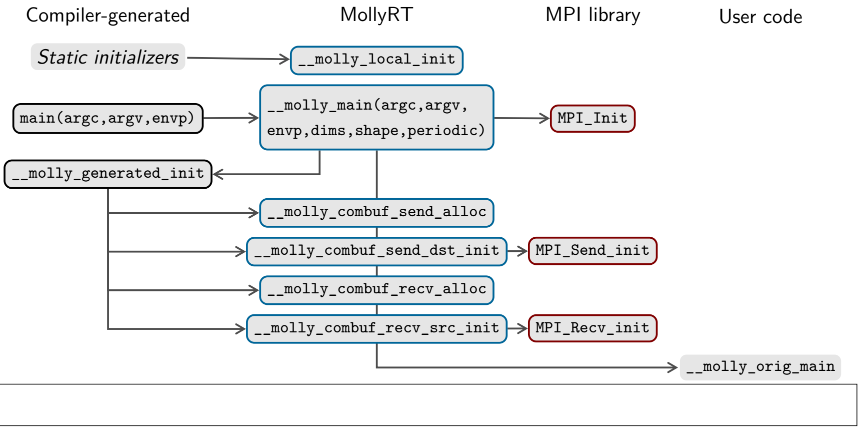
MollyRT

Init- and finalization



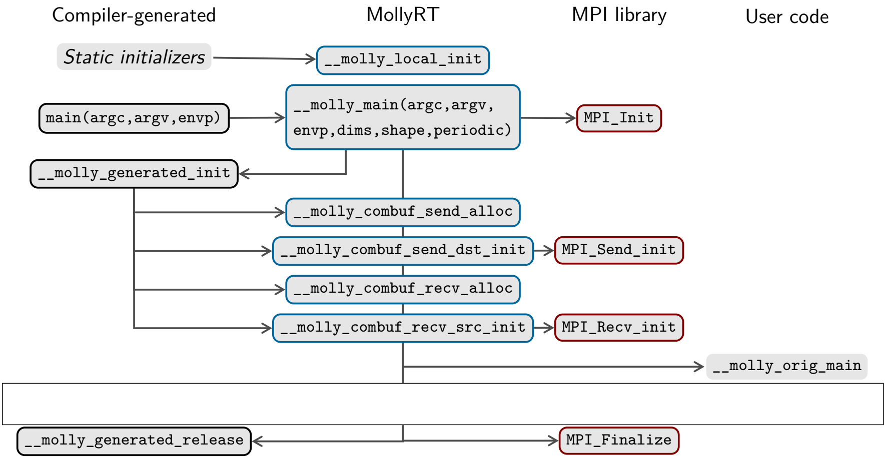
MollyRT

Init- and finalization



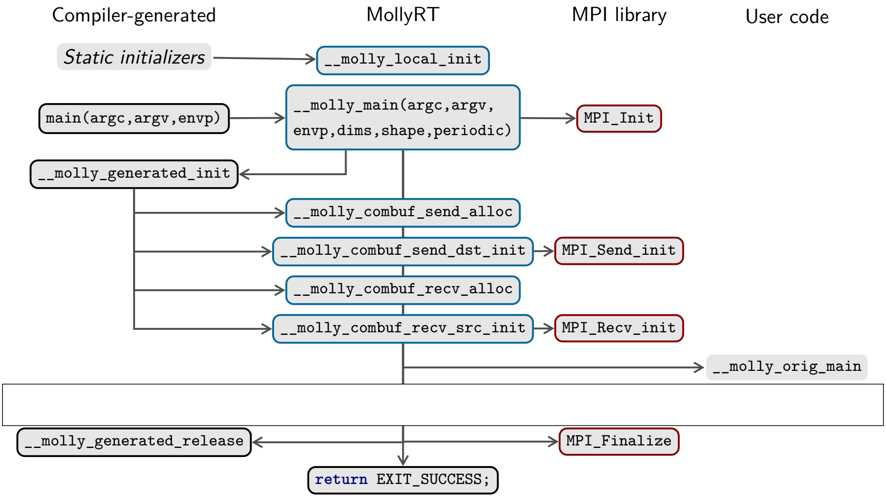
MollyRT

Init- and finalization



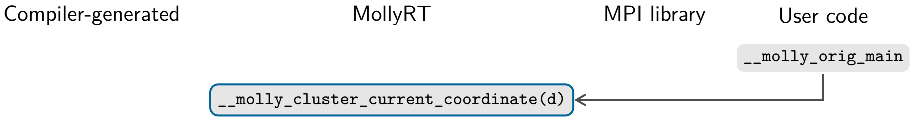
MollyRT

Init- and finalization



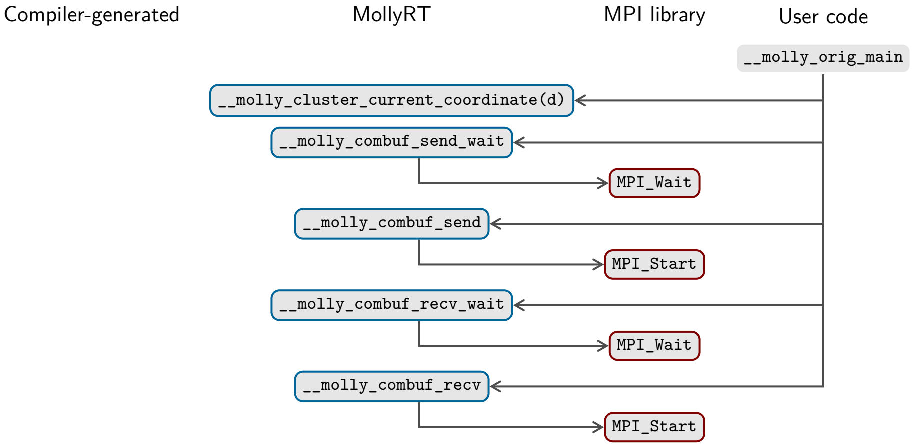
MollyRT

User code

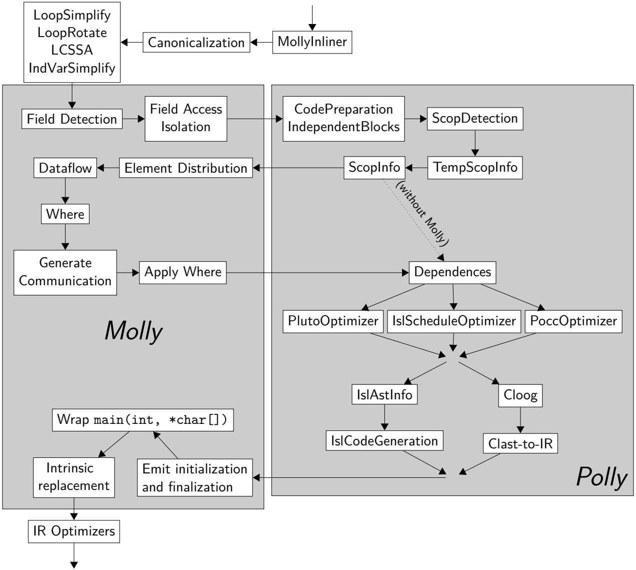


MollyRT

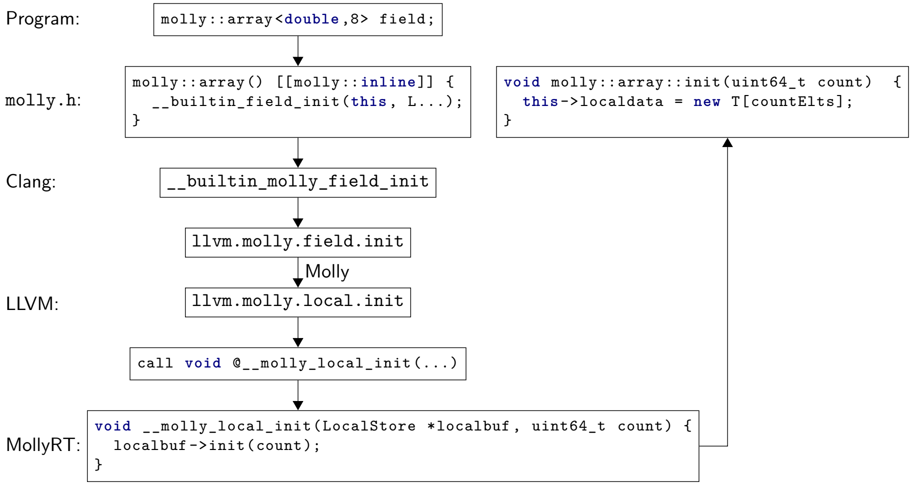
User code



Molly Pipeline Close-up



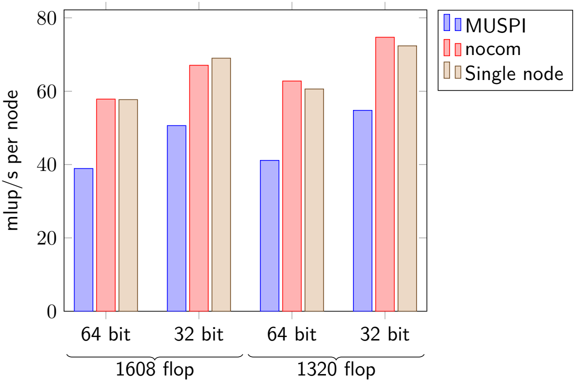
Transformation Process



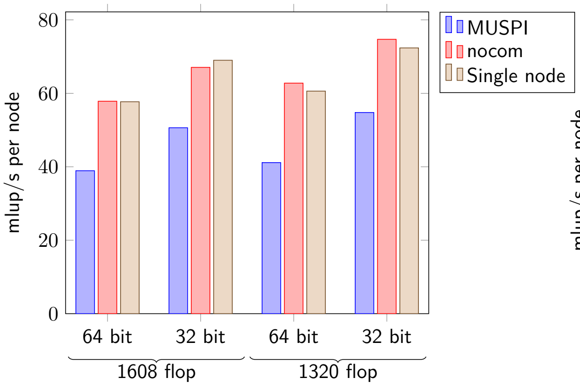
Dslash Source for Molly

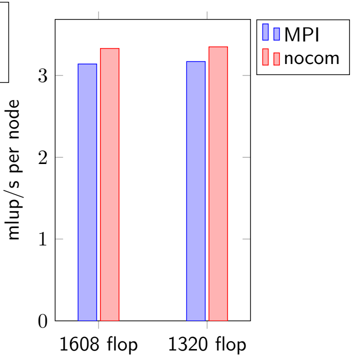
```
#include <molly.h>
#include <lqcd.h>

#pragma molly transform("{ [t,x,y,z] -> [node[floor(t/12),floor(x/12),floor(y/12),floor(z/12)] -> local[
    floor(t/2),x,y,z,t%2]] }")
molly::array<spinor_t, 48, 24, 24, 24> source, sink;

#pragma molly transform("{ [t,x,y,z,d] -> [node[pt,px,py,pz] -> local[t,x,y,z,d]] : 0<=pt<4 and 0<=px<2
    and 0<=py<2 and 0<=pz<2 and 12pt<=t<=12*(pt+1) and 12px<=x<=12*(px+1) and 12py<=y<=12*(py+1) and 12pz
    <=z<12*(pz+1) }")
molly::array<su3matrix_t, 48 + 1, 24 + 1, 24 + 1, 24 + 1, 4> gauge;

void HoppingMatrix() {
  for (int t = 0; t < source.length(0); t += 1)
    for (int x = 0; x < source.length(1); x += 1)
      for (int y = 0; y < source.length(2); y += 1)
        for (int z = 0; z < source.length(3); z += 1) {
          auto halfspinor = project_TUP(source[molly::mod(t + 1, LT)][x][y][z]);
          halfspinor = gauge[t + 1][x][y][z][DIM_T] * halfspinor;
          auto result = expand_TUP(halfspinor);

          halfspinor = project_TDN(source[molly::mod(t - 1, LT)][x][y][z]);
          halfspinor = gauge[t][x][y][z][DIM_T] * halfspinor;
          result += expand_TDN(halfspinor);

          // [...]

          sink[t][x][y][z] = result;
        }
}
```

Benchmark Results



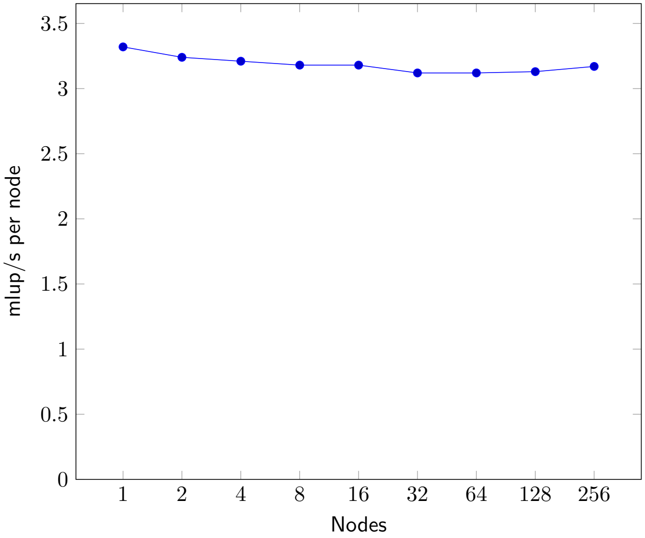
Benchmark Results





Benchmark Results

Weak Scaling



Discussion

Conclusions

- **Manual implementation** (used eg. by Nessi library)
  - Up to **55% of peak** performance, 16000 lines of code
- Current achievement of **Molly**
  - 2.5% of peak performance, **50 lines** of code (+ library)

Conclusions

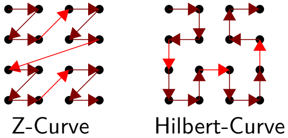
- **Manual implementation** (used eg. by Nessi library)
  - Up to **55% of peak** performance, 16000 lines of code
- Current achievement of **Molly**
  - 2.5% of peak performance, **50 lines** of code (+ library)

- Molly can be competitive in the future!
  - Keep $\mathbb{Z}$-polytope complexity low
  - HPC-specific LLVM passes (eg. prefetching, strength reduction)
  - On-the-fly optimization
  - Usability

Conclusions

- **Manual implementation** (used eg. by Nessi library)
  - Up to **55% of peak** performance, 16000 lines of code
- Current achievement of **Molly**
  - 2.5% of peak performance, **50 lines** of code (+ library)

- Molly can be competitive in the future!
  - Keep $\mathbb{Z}$-polytope complexity low
  - HPC-specific LLVM passes (eg. prefetching, strength reduction)
  - On-the-fly optimization
  - Usability
- Let's have a framework on memory transformations
  - Field autodetection
  - First class `molly::array`/element type
  - **Dynamically**-sized arrays
  - Dynamic memory **layouts**




That's all Folks!