PIPES: A Language and Compiler for Task-based Programming on Distributed-Memory Clusters

Martin Kong[1,2] Louis-Noël Pouchet[3,2] P. Sadayappan[2] Vivek Sarkar[1]

[1] **Rice University**
[2] **The Ohio State University**
[3] **Colorado State University**

September 2016
8th Concurrent Collections Workshop
Rochester, NY

Structure of this talk

1. Motivation
2. PIPES Language
3. PIPES Compiler
4. Some performance results
5. Conclusion

Structure of this talk

1. Motivation
2. PIPES Language
3. PIPES Compiler
4. Some performance results
5. Conclusion

Structure of this talk

1. Motivation
2. PIPES Language
3. PIPES Compiler
4. Some performance results
5. Conclusion

Structure of this talk

1. Motivation
2. PIPES Language
3. PIPES Compiler
4. Some performance results
5. Conclusion

Structure of this talk

1. Motivation
2. PIPES Language
3. PIPES Compiler
4. Some performance results
5. Conclusion

Context of this work

- We target clusters of shared-memory computers using Intel CnC C++
- MPI defacto standard for distributed-memory computing
- Can achieve high-performance
- Writing MPI code is:
  - Error prone
  - Hard to debug and maintain

Context of this work

- We target clusters of shared-memory computers using Intel CnC C++
- MPI defacto standard for distributed-memory computing
- Can achieve high-performance
- Writing MPI code is:
  - Error prone
  - Hard to debug and maintain

Context of this work

- We target clusters of shared-memory computers using Intel CnC C++
- MPI defacto standard for distributed-memory computing
- Can achieve high-performance
- Writing MPI code is:
  - Error prone
  - Hard to debug and maintain

Context of this work

- We target clusters of shared-memory computers using Intel CnC C++
- MPI defacto standard for distributed-memory computing
- Can achieve high-performance
- Writing MPI code is:
  - Error prone
  - Hard to debug and maintain

Context of this work

- We target clusters of shared-memory computers using Intel CnC C++
- MPI defacto standard for distributed-memory computing
- Can achieve high-performance
- Writing MPI code is:
  - Error prone
  - Hard to debug and maintain

Context of this work

- We target clusters of shared-memory computers using Intel CnC C++
- MPI defacto standard for distributed-memory computing
- Can achieve high-performance
- Writing MPI code is:
  - Error prone
  - Hard to debug and maintain

Programming System: our objectives

- Powerful and flexible run-time for distributed computing (Intel CnC C++)
- High productivity system (compact/expressive language, high-performance)
- Automatic analyses and transformations on the dataflow graph (i.e. coarsening and coalescing)
- Systematic generation of various program variants (distribution, communication, scheduling)
- Separate algorithmic from performance specification

Programming System: our objectives

- Powerful and flexible run-time for distributed computing (Intel CnC C++)
- High productivity system (compact/expressive language, high-performance)
- Automatic analyses and transformations on the dataflow graph (i.e. coarsening and coalescing)
- Systematic generation of various program variants (distribution, communication, scheduling)
- Separate algorithmic from performance specification

Programming System: our objectives

- Powerful and flexible run-time for distributed computing (Intel CnC C++)
- High productivity system (compact/expressive language, high-performance)
- Automatic analyses and transformations on the dataflow graph (i.e. coarsening and coalescing)
- Systematic generation of various program variants (distribution, communication, scheduling)
- Separate algorithmic from performance specification

Programming System: our objectives

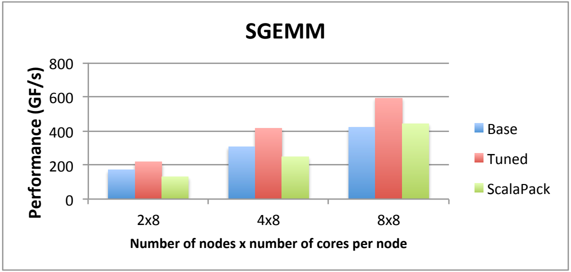
- Powerful and flexible run-time for distributed computing (Intel CnC C++)
- High productivity system (compact/expressive language, high-performance)
- Automatic analyses and transformations on the dataflow graph (i.e. coarsening and coalescing)
- Systematic generation of various program variants (distribution, communication, scheduling)
- Separate algorithmic from performance specification

Programming System: our objectives

- Powerful and flexible run-time for distributed computing (Intel CnC C++)
- High productivity system (compact/expressive language, high-performance)
- Automatic analyses and transformations on the dataflow graph (i.e. coarsening and coalescing)
- Systematic generation of various program variants (distribution, communication, scheduling)
- Separate algorithmic from performance specification

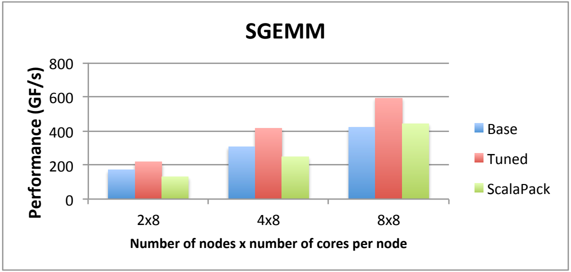
Motivating example: SGEMM

- Two variants: Cannon (2D parallel, lockstep), Johnson (3D parallel + reduction phase)
- PIPES input: +/- 35 lines of code
- PIPES output: 800-1200 lines
- Similarity of Johnson and Cannon PIPES variants > 60%
- Achieve above 50% of machine peak on 8 nodes (4-core dual-socket)
- Intel MKL for task bodies



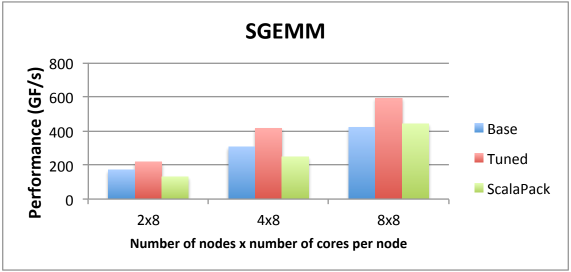
Motivating example: SGEMM

- Two variants: Cannon (2D parallel, lockstep), Johnson (3D parallel + reduction phase)
- PIPES input: +/- 35 lines of code
- PIPES output: 800-1200 lines
- Similarity of Johnson and Cannon PIPES variants > 60%
- Achieve above 50% of machine peak on 8 nodes (4-core dual-socket)
- Intel MKL for task bodies



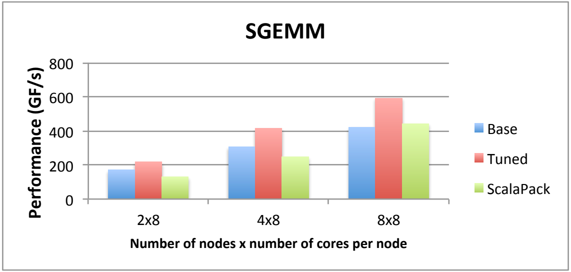
Motivating example: SGEMM

- Two variants: Cannon (2D parallel, lockstep), Johnson (3D parallel + reduction phase)
- PIPES input: +/- 35 lines of code
- PIPES output: 800-1200 lines
- Similarity of Johnson and Cannon PIPES variants > 60%
- Achieve above 50% of machine peak on 8 nodes (4-core dual-socket)
- Intel MKL for task bodies



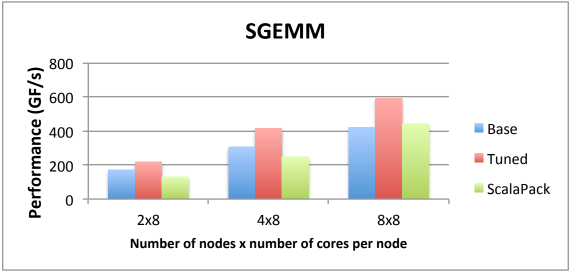
Motivating example: SGEMM

- Two variants: Cannon (2D parallel, lockstep), Johnson (3D parallel + reduction phase)
- PIPES input: +/- 35 lines of code
- PIPES output: 800-1200 lines
- Similarity of Johnson and Cannon PIPES variants > 60%
- Achieve above 50% of machine peak on 8 nodes (4-core dual-socket)
- Intel MKL for task bodies



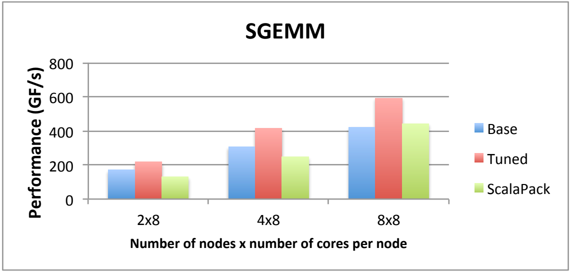
Motivating example: SGEMM

- Two variants: Cannon (2D parallel, lockstep), Johnson (3D parallel + reduction phase)
- PIPES input: +/- 35 lines of code
- PIPES output: 800-1200 lines
- Similarity of Johnson and Cannon PIPES variants > 60%
- Achieve above 50% of machine peak on 8 nodes (4-core dual-socket)
- Intel MKL for task bodies



Motivating example: SGEMM

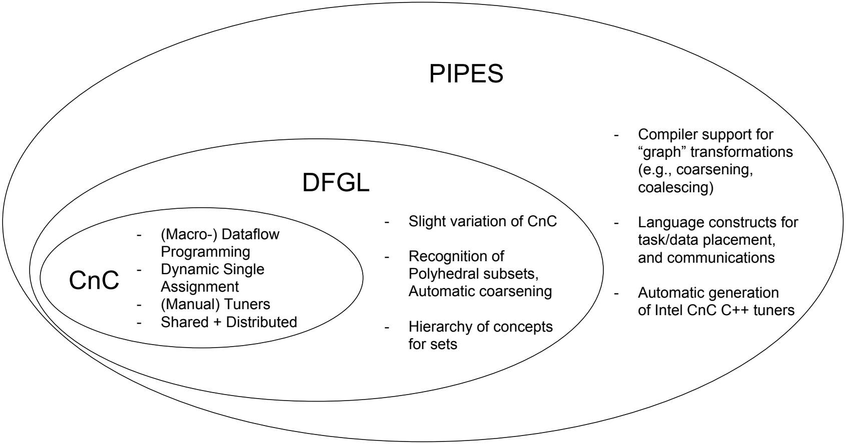
- Two variants: Cannon (2D parallel, lockstep), Johnson (3D parallel + reduction phase)
- PIPES input: +/- 35 lines of code
- PIPES output: 800-1200 lines
- Similarity of Johnson and Cannon PIPES variants > 60%
- Achieve above 50% of machine peak on 8 nodes (4-core dual-socket)
- Intel MKL for task bodies



Related Work

1. Burke, Knobe, Newton and Sarkar, "Concurrent Collections Programming Model", Encyclopedia of Parallel Computing, 2010
   - Data-flow run-time model
   - No automatic or manual tuning capabilities
2. Knobe and Burke, "The Tuning Language for Concurrent Collections", CPC, 2012
   - Hierarchical affinity groups
   - No Automatic code generation, user annotated
3. Schlimbach, Brodman, Knobe, "Concurrent Collections on Distributed Memory: theory put into practice", EuroMICRO 2013
   - (Manual) Tuner capabilities
   - Distributed CnC
4. Sbirlea et al., "DFGR: An Intermediate Graph Representation for Macro-Dataflow Programs", DFM, 2014
   - Ranges, polyhedra, union of polyhedra
   - No explicit modeling of task/data placement and communications

Contributions



Core PIPES Language Features

- Virtual topologies
- Task placement
- Data placement
- Data communication (pull or pull communication model)

DFGL MatMul

```

// Define data collections
[float* A:1..N,1..N];
...
// Task prescriptions
env :: (MM:1..N,1..N,1..N);
// Input/Output:
env -> [A:1..N,1..N];
...
[C:1..N,1..N,N] -> env;
// Task dataflow
[A:i,k],[B:k,j],[C:i,j,k] -> (MM:i,j,k) -> [C:i,j,k+1];

//

//


//
```

Figure: DFGL Matrix Multiplication

PIPES Cannon

```
Parameter N, P;
// Define data collections
[float* A:1..N,1..N];
...
// Task prescriptions
env :: (MM:1..N,1..N,1..N);
// Input/Output:
env -> [A:1..N,1..N];
...
[C:1..N,1..N,N] -> env;
// Task dataflow
[A:i,k],[B:k,j],[C:i,j,k] -> (MM:i,j,k) -> [C:i,j,k+1];
Topology Proc = Topo2D(P,P);
// Place the N tasks (i,j,*) to Proc((i/8)%P,(j/8)%P)
(MM:i,j,1..N)@Proc((i/8)%P,(j/8)%P);
// Circular communication pattern for Cannon algorithm
[A:i,k]@(MM:i,j,k) => (MM:i,(j-1)%P,k+1);
[B:k,j]@(MM:i,j,k) => (MM:(i-1)%P,j,k+1);
// end
```

Figure: PIPES Cannon Matrix Multiplication

Virtual Topologies

- Represents the logical underlying computer grid/cluster
- Each element in the set is a processor
- Requires a logical-to-physical mapping

```
// 2D topology, no more than 256x256 processors
Parameter P : 1..256;
Topology Topo2D = {
  sizes=[P,P];
  cores=[i,j] : { 0 <= i < P, 0 <= j < P};
};
```

Virtual Topologies

- Represents the logical underlying computer grid/cluster
- Each element in the set is a processor
- Requires a logical-to-physical mapping

```
// 2D topology, no more than 256x256 processors
Parameter P : 1..256;
Topology Topo2D = {
  sizes=[P,P];
  cores=[i,j] : { 0 <= i < P, 0 <= j < P};
};
```

Virtual Topologies

- Represents the logical underlying computer grid/cluster
- Each element in the set is a processor
- Requires a logical-to-physical mapping

```
// 2D topology, no more than 256x256 processors
Parameter P : 1..256;
Topology Topo2D = {
  sizes=[P,P];
  cores=[i,j] : { 0 <= i < P, 0 <= j < P};
};
```

Task Placement

- Mappings of tasks to elements in the topology
- Task (instance) will execute on the processor it is mapped to
- Always enforced by run-time
- Requires the topology to be defined
- Maps directly to the **compute_on** tuner

```
(task : tag-set) @ TopologyId(point);
```

Task Placement

- Mappings of tasks to elements in the topology
- Task (instance) will execute on the processor it is mapped to
- Always enforced by run-time
- Requires the topology to be defined
- Maps directly to the **compute_on** tuner

```
(task : tag-set) @ TopologyId(point);
```

Task Placement

- Mappings of tasks to elements in the topology
- Task (instance) will execute on the processor it is mapped to
- Always enforced by run-time
- Requires the topology to be defined
- Maps directly to the **compute_on** tuner

```
(task : tag-set) @ TopologyId(point);
```

Task Placement

- Mappings of tasks to elements in the topology
- Task (instance) will execute on the processor it is mapped to
- Always enforced by run-time
- Requires the topology to be defined
- Maps directly to the **compute_on** tuner

```
(task : tag-set) @ TopologyId(point);
```

Task Placement

- Mappings of tasks to elements in the topology
- Task (instance) will execute on the processor it is mapped to
- Always enforced by run-time
- Requires the topology to be defined
- Maps directly to the **compute_on** tuner

```
(task : tag-set) @ TopologyId(point);
```

Data Communication

- Meant for enforcing a communication order
- Use tasks as referentials
- LHS of '=>' specifies an owner of the data
- RHS of '=>' specifies the new consumer
- Owner of data can be:
  - original producer or
  - one of its explicits consumers

```
[item: tag-set] @ (task1 : tag) => (task2 : tag);
```

Data Communication

- Meant for enforcing a communication order
- Use tasks as referentials
- LHS of '=>' specifies an owner of the data
- RHS of '=>' specifies the new consumer
- Owner of data can be:
  - original producer or
  - one of its explicits consumers

```
[item: tag-set] @ (task1 : tag) => (task2 : tag);
```

Data Communication

- Meant for enforcing a communication order
- Use tasks as referentials
- LHS of '=>' specifies an owner of the data
- RHS of '=>' specifies the new consumer
- Owner of data can be:
  - original producer or
  - one of its explicits consumers

```
[item: tag-set] @ (task1 : tag) => (task2 : tag);
```

Data Communication

- Meant for enforcing a communication order
- Use tasks as referentials
- LHS of '=>' specifies an owner of the data
- RHS of '=>' specifies the new consumer
- Owner of data can be:
  - original producer or
  - one of its explicits consumers

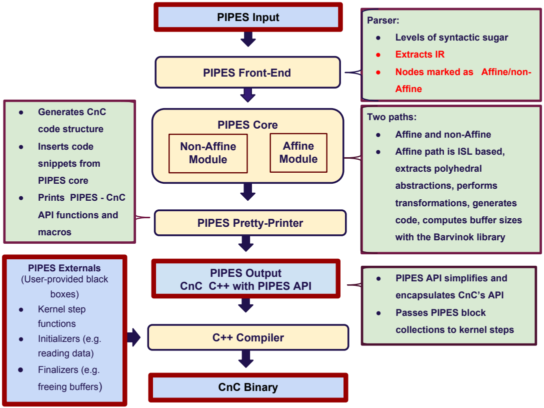
```
[item: tag-set] @ (task1 : tag) => (task2 : tag);
```

Data Communication

- Meant for enforcing a communication order
- Use tasks as referentials
- LHS of '=>' specifies an owner of the data
- RHS of '=>' specifies the new consumer
- Owner of data can be:
  - original producer or
  - one of its explicits consumers

```
[item: tag-set] @ (task1 : tag) => (task2 : tag);
```

Compiler Framework



PIPES Compiler Analyses and Transformations

- Recognition of polyhedral subset of PIPES
- Translate (sub-)graph to polyhedral representation (iteration domains, dataflow relations)
- Use off-the-shelf polyhedral optimization tools (PLuTo and ISL to perform transformations)
- Implement transformation at the graph level
  - Iteration domains of tile loops become task sets
  - Dataflow relation remains unchanged; data is not coarsened

Coarsening in PIPES

- Automatic coarsening (tiling) of task space
- Use a slightly modified version of the **Tiling-Hyperplane Method** (Bondhugula, PLDI'08) to maximize tilability opportunities

$$-\vec{u}.\vec{n} - w \leq \Theta_R(\vec{x}_R) - \Theta_S(\vec{x}_S) \leq \vec{u}.\vec{n} + w$$

- High potential to minimize run-time overhead

```
...
// Task prescriptions
env :: (mul:1..N,1..N,1..N);
env :: (add:1..N,1..N,1..N);
// Tiling by 100x100x100 =>
// Task space = [0..N/100-1,0..N/100-1,0..N/100-1]
```

Coalescing in PIPES

- Fusion of tasks
- Data from producer-consumer tasks becomes intra-task communication
- User can explicitly fuse tasks

```
...
// Task prescriptions
env :: (mul:1..N,1..N,1..N);
env :: (add:1..N,1..N,1..N);
// Task dataflow
[A:i,k],[B:k,j] -> (mul:i,j,k) -> [C:i,j,k];
[C:i,j,k],[D:i,j,k] -> (add:i,j,k) -> [D:i,j,k+1];
coalesce ([mul:i,j,k],[add:i,j,k]);
// result:
// [A:i,k],[B:k,j],[D:i,j,k] -> (muladd:i,j,k) -> [D:i,j,k+1];
```

CnC Tuners

We leverage the following CnC tuners

- Compute On
- Consumed On
- Produced On
- Dependency Consumer
- Get Count

CnC Tuners

We leverage the following CnC tuners

- Compute On
- Consumed On
- Produced On
- Dependency Consumer
- Get Count

CnC Tuners

We leverage the following CnC tuners

- Compute On
- Consumed On
- Produced On
- Dependency Consumer
- Get Count

CnC Tuners

We leverage the following CnC tuners

- Compute On
- Consumed On
- Produced On
- Dependency Consumer
- Get Count

CnC Tuners

We leverage the following CnC tuners

- Compute On
- Consumed On
- Produced On
- Dependency Consumer
- Get Count

Compute On Tuner

- Tuner associates a processor rank to each task instance
- Requires a logical-to-physical mapping (user provided or user selected via compiler option)
- User must provide an affinity map (i.e. task @ topo)
- For each task, apply affinity map to task iteration domain
- Generated statement is a call to the logical-to-physical mapping with the task tuple as argument

Compute On Tuner

- Tuner associates a processor rank to each task instance
- Requires a logical-to-physical mapping (user provided or user selected via compiler option)
- User must provide an affinity map (i.e. task @ topo)
- For each task, apply affinity map to task iteration domain
- Generated statement is a call to the logical-to-physical mapping with the task tuple as argument

Compute On Tuner

- Tuner associates a processor rank to each task instance
- Requires a logical-to-physical mapping (user provided or user selected via compiler option)
- User must provide an affinity map (i.e. task @ topo)
- For each task, apply affinity map to task iteration domain
- Generated statement is a call to the logical-to-physical mapping with the task tuple as argument

Compute On Tuner

- Tuner associates a processor rank to each task instance
- Requires a logical-to-physical mapping (user provided or user selected via compiler option)
- User must provide an affinity map (i.e. task @ topo)
- For each task, apply affinity map to task iteration domain
- Generated statement is a call to the logical-to-physical mapping with the task tuple as argument

Compute On Tuner

- Tuner associates a processor rank to each task instance
- Requires a logical-to-physical mapping (user provided or user selected via compiler option)
- User must provide an affinity map (i.e. task @ topo)
- For each task, apply affinity map to task iteration domain
- Generated statement is a call to the logical-to-physical mapping with the task tuple as argument

Compute On Tuner

```
PIPES_TASK_TUNER_AFFINITY_HEADER(cannon,MMC) {
  // Insert LOG2PHYS function call here
  int kk0;
  int kk1;
  int kk2;
  kk0 = local_tag[0];
  kk1 = local_tag[1];
  kk2 = local_tag[2];
  // Convert task tuple to topology tuple
  int pos0 = PIPES_MAP_AFFINITY_MMC_TO_G_0(N, P1, P2, TS, TUB,
      kk0, kk1, kk2) ;
  int pos1 = PIPES_MAP_AFFINITY_MMC_TO_G_1(N, P1, P2, TS, TUB,
      kk0, kk1, kk2) ;
  int ret;
  ret = pipes_log2phy (N, P1, P2, TS, TUB, pos0, pos1);
  return ret;
}
```

Consumed On Tuner

- Automatically determines the processor rank on which a data instance is consumed
- Enforces a push communication model (initiated by producer)
- Allows point-to-point communication
- Support for multiple consumers

---

for each block collection b **do**
    rank_consumers = $\varnothing$
    for each producer relation $p \rightarrow b$ **do**
        for each consumer relation $b \rightarrow c$ **do**
            task_consumer = $(p \rightarrow b) \circ (b \rightarrow c)$
            affinity_map = find_affinity_map ($IMG(task\_consumer)$)
            rank_consumers += $IMG(affinity\_map)$
        end for
    end for
    codegen (b, rank_consumers, consumed_on)
end for

Consumed On Tuner

- Automatically determines the processor rank on which a data instance is consumed
- Enforces a push communication model (initiated by producer)
- Allows point-to-point communication
- Support for multiple consumers

for each block collection b **do**
  rank_consumers = $\varnothing$
  for each producer relation $p \rightarrow b$ **do**
    for each consumer relation $b \rightarrow c$ **do**
      task_consumer = $(p \rightarrow b) \circ (b \rightarrow c)$
      affinity_map = find_affinity_map ($IMG(task\_consumer)$)
      rank_consumers += $IMG(affinity\_map)$
    end for
  end for
  codegen (b, rank_consumers, consumed_on)
end for

Consumed On Tuner

- Automatically determines the processor rank on which a data instance is consumed
- Enforces a push communication model (initiated by producer)
- Allows point-to-point communication
- Support for multiple consumers

---

for each block collection b **do**
  rank_consumers = $\varnothing$
  for each producer relation $p \rightarrow b$ **do**
    for each consumer relation $b \rightarrow c$ **do**
      task_consumer = $(p \rightarrow b) \circ (b \rightarrow c)$
      affinity_map = find_affinity_map ($IMG(task\_consumer)$)
      rank_consumers += $IMG(affinity\_map)$
    end for
  end for
  codegen (b, rank_consumers, consumed_on)
end for

Consumed On Tuner

- Automatically determines the processor rank on which a data instance is consumed
- Enforces a push communication model (initiated by producer)
- Allows point-to-point communication
- Support for multiple consumers

---

for each block collection b **do**
  rank_consumers = $\varnothing$
  for each producer relation $p \rightarrow b$ **do**
    for each consumer relation $b \rightarrow c$ **do**
      task_consumer = $(p \rightarrow b) \circ (b \rightarrow c)$
      affinity_map = find_affinity_map ($IMG(task\_consumer)$)
      rank_consumers += $IMG(affinity\_map)$
    end for
  end for
  codegen (b, rank_consumers, consumed_on)
end for

Consumed On Tuner

```
if (kk0 >= 0 && N >= 1000 * kk0 + 1 && kk1 >= 0 && N >= 1000 *
    kk1 + 1 && N >= 1000 * kk2 && 1000 * kk2 + 999 >= N) {
    // non-local push placement for env
    dim0 = PIPES_MAP_AFFINITY_env_TO_G_0(N, P1, P2, TS, TUB,0 )
        ;
    dim1 = PIPES_MAP_AFFINITY_env_TO_G_1(N, P1, P2, TS, TUB,0 )
        ;
    _r = pipes_log2phy (N, P1, P2, TS, TUB, dim0, dim1);
    rank_set.insert (_r);
  }
  if (kk0 >= 0 && N >= 1000 * kk0 + 1 && kk1 >= 0 && N >= 1000 *
       kk1 + 1 && kk2 >= 0 && N >= 1000 * kk2 + 1) {
    // non-local push placement for MMC
    dim0 = PIPES_MAP_AFFINITY_MMC_TO_G_0(N, P1, P2, TS, TUB,kk0
        ,kk1 ,kk2 ) ;
    dim1 = PIPES_MAP_AFFINITY_MMC_TO_G_1(N, P1, P2, TS, TUB,kk0
        ,kk1 ,kk2 ) ;
    _r = pipes_log2phy (N, P1, P2, TS, TUB, dim0, dim1);
    rank_set.insert (_r);
  }
}
```

Produced On Tuner

- Automatically determines the processor rank on which a data instance is produced
- Enforces a pull communication model (initiated by consumer)
- Allows point-to-point communication
- Always single producer (by DSA property)

---

for each block collection b **do**
rank_producers = $\varnothing$
for each consumer relation $b \rightarrow c$ **do**
for each producer relation $p \rightarrow b$ **do**
task_producer = $(b \rightarrow c)^{-1} \circ (p \rightarrow b)^{-1}$
affinity_map = find_affinity_map ($IMG(task\_producer)$)
rank_producers += $IMG(affinity\_map)$
end for
end for
codegen (b, rank_producers, produced_on)
end for

Produced On Tuner

- Automatically determines the processor rank on which a data instance is produced
- Enforces a pull communication model (initiated by consumer)
- Allows point-to-point communication
- Always single producer (by DSA property)

---

for each block collection b **do**
  rank_producers = $\varnothing$
  for each consumer relation $b \rightarrow c$ **do**
    for each producer relation $p \rightarrow b$ **do**
      task_producer = $(b \rightarrow c)^{-1} \circ (p \rightarrow b)^{-1}$
      affinity_map = find_affinity_map ($IMG(task\_producer)$)
      rank_producers += $IMG(affinity\_map)$
    end for
  end for
  codegen (b, rank_producers, produced_on)
end for

Produced On Tuner

- Automatically determines the processor rank on which a data instance is produced
- Enforces a pull communication model (initiated by consumer)
- Allows point-to-point communication
- Always single producer (by DSA property)

for each block collection b **do**
rank_producers = $\emptyset$
for each consumer relation $b \to c$ **do**
for each producer relation $p \to b$ **do**
task_producer = $(b \to c)^{-1} \circ (p \to b)^{-1}$
affinity_map = find_affinity_map ($IMG(task\_producer)$)
rank_producers += $IMG(affinity\_map)$
end for
end for
codegen (b, rank_producers, produced_on)
end for

Produced On Tuner

- Automatically determines the processor rank on which a data instance is produced
- Enforces a pull communication model (initiated by consumer)
- Allows point-to-point communication
- Always single producer (by DSA property)

---

for each block collection b **do**
  rank_producers = $\varnothing$
  for each consumer relation $b \rightarrow c$ **do**
    for each producer relation $p \rightarrow b$ **do**
      task_producer = $(b \rightarrow c)^{-1} \circ (p \rightarrow b)^{-1}$
      affinity_map = find_affinity_map ($IMG(task\_producer)$)
      rank_producers += $IMG(affinity\_map)$
    end for
  end for
  codegen (b, rank_producers, produced_on)
end for

Dependency Consumer Tuner

- Allows to specify the exact data instances on which a task instance depends
- If not defined, run-time is free to schedule tasks in any order

---

for each task collection t **do**
    deps = $\varnothing$
    for each consumer relation $b \rightarrow t$ **do**
        $(t \rightarrow b) = (b \rightarrow t)^{-1}$
        Convert input tuple dimensions of $(t \rightarrow b)$ to parameters
        deps += $IMG(t \rightarrow b)$
    end for
    codegen (t, deps, dependency_consumer)
end for

Dependency Consumer Tuner

- Allows to specify the exact data instances on which a task instance depends
- If not defined, run-time is free to schedule tasks in any order

---

for each task collection t **do**
  deps = $\varnothing$
  for each consumer relation $b \rightarrow t$ **do**
    $(t \rightarrow b) = (b \rightarrow t)^{-1}$
    Convert input tuple dimensions of $(t \rightarrow b)$ to parameters
    deps += $IMG(t \rightarrow b)$
  end for
  codegen (t, deps, dependency_consumer)
end for

Get Count Tuner

- Determines the number of times a block instance is read
- Leverages the Barvinok Library to count points (Verdoolaege et al., Algorithmica, 2007)

for each block collection b **do**
    for each consumer relation $b \rightarrow c$ **do**
        count = 0
        Convert input tuple dimensions of $b \rightarrow c$ to parameters
        for each task t in $IMG(b \rightarrow c)$ **do**
            count += $CARD(t)$
        end for
    end for
    codegen (b, count, maxlife)
end for

Get Count Tuner

- Determines the number of times a block instance is read
- Leverages the Barvinok Library to count points (Verdoolaege et al., Algorithmica, 2007)

---

for each block collection b **do**
    for each consumer relation $b \to c$ **do**
        count = 0
        Convert input tuple dimensions of $b \to c$ to parameters
        for each task t in $IMG(b \to c)$ **do**
            count += $CARD(t)$
        end for
    end for
    codegen (b, count, maxlife)
end for

Experimental Setup

Parameters	Value
Nodes	1-8
Processor	Intel Xeon E5630 @ 2.5 GHz
Sockets per node	2
Cores per socket	4
Intra-node bandwidth	25000 MB/s
InfiniBand QDR bandwidth	5120 MB/s
L1 Cache	32 KB per core
L2 Cache	256 KB per core
L3 Cache	12 MB per socket
Intel CnC C++	1.01
MPI run-time	Intel MPI 5.0
Compiler	Intel ICPC 13
Slurm	2.6.5

Table: Experimental setup

SSYR2K

The Symmetric rank-2 update computes the function:

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      C[i][j] += A[j][k]*alpha*B[i][k]+B[j][k]*alpha*A[i][k]
```

SSYR2K

The Symmetric rank-2 update computes the function:

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      C[i][j] += B[i][k]*A[j][k]*alpha

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      C[i][j] += A[i][k]*B[j][k]*alpha
```

So we can represent this computation equivalently with two

GEMM(C, B, trans(A))

GEMM(C, A, trans(B))

SSYR2K Variants

Parallel variant exploits $SGEMM(C, A, trans(B))$

```
[C:i,j,k], [A:i,k], [B:j,k] -> (GEMM:i,j,k,0) -> [C:i,j,k+1];
[D:i,j,k], [B:i,k], [A:j,k] -> (GEMM:i,j,k,1) -> [D:i,j,k+1];
[C:i,j,N], [D:j,i,N] -> (AddMat:i,j) -> [Res:i,j];
```

Figure: Parallel SYR2K

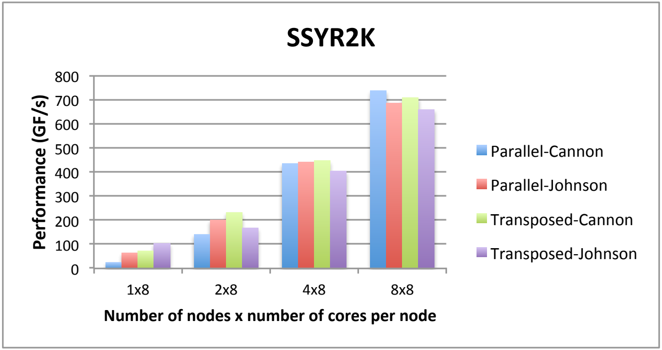
Transposed variant exploits:

$$trans(SGEMM(trans(A), trans(B))) = SGEMM(A, B)$$

```
[C:i,j,k], [A:i,k], [B:j,k] ->(GEMM:i,j,k,0) -> [C:i,j,k+1];
[D:i,j,k], [B:j,k], [A:i,k] ->(GEMM:i,j,k,1) -> [D:i,j,k+1];
[C:i,j,N], [D:j,i,N] -> (AddMat:i,j) -> [Res:i,j];
```

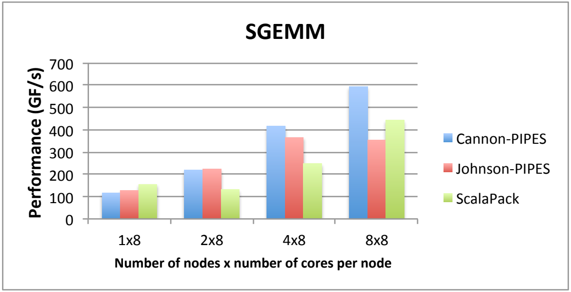
Figure: Transposed SYR2K

SSYR2K



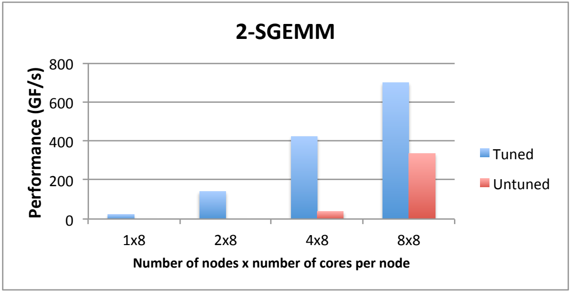
- PIPES inputs: 40-55 lines; generated code: 1000-1700 lines
- System allows to explore/test different classical and new algorithms

SGEMM



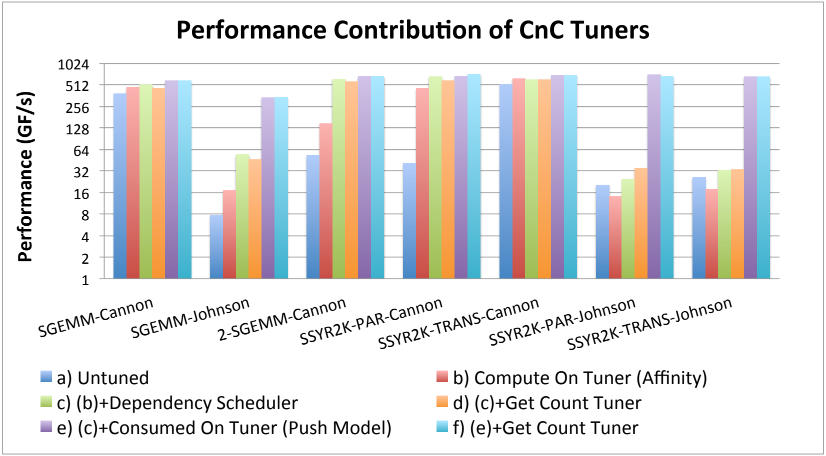
- Cannon and Johnson Algorithms achieve near/above 50% machine peak
- High productivity framework

2MM



- Automatically generated tuners: key for high-performance
- Composed 2 PIPES Cannon

Performance Breakdown



PIPES Statistics

Variant	Input Lines	Output Lines
SGEMM Cannon	40	1050
SGEMM Johnson	30	900
SSYR2K Parallel Cannon	55	1700
SSYR2K Trasposed Cannon	45	1400
SSYR2K Parallel Johnson	40	1200
SSYR2K Transposed Johnson	40	1000

Table: Line Stats

Take Home Message

Key problem: productivity+performance on distributed clusters?

Current state of practice:

- MPI: difficult/tedious to write, but can deliver high performance
- CnC: high-level dependence specification, but performance still hard to obtain
- What is needed: a **compiler framework** for CnC to help generating high-performance code.

PIPES brings productivity, performance demonstrated for several codes:

- Enables explicit description of parallel algorithm specifications
- Leverage work on CnC and DFGL, target Intel CnC C++
- Automatic graph analysis and transformations
- Automatic CnC tuner generation for high-performacne

Ongoing and future work

- Task placement, single (Intel CnC C++) and multi-level (HCLib++)
- Graph composition and reusability
- Overhead reduction of CnC programs
- Task isolation
- Collectives optimization

Ongoing and future work

C'est fini