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

Martin Kong\textsuperscript{1,2} Louis-Noël Pouchet\textsuperscript{3,2} P. Sadayappan\textsuperscript{2} Vivek Sarkar\textsuperscript{1}

\textsuperscript{1} Rice University
\textsuperscript{2} The Ohio State University
\textsuperscript{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
Motivation:
CnC'16
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

<table>
<thead>
<tr>
<th>Number of nodes x number of cores per node</th>
<th>Base</th>
<th>Tuned</th>
<th>ScalaPack</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x8</td>
<td>0</td>
<td>200</td>
<td>300</td>
</tr>
<tr>
<td>4x8</td>
<td>400</td>
<td>600</td>
<td>500</td>
</tr>
<tr>
<td>8x8</td>
<td>800</td>
<td>1200</td>
<td>1000</td>
</tr>
</tbody>
</table>

![Graph showing performance (GF/s) for SGEMM](chart.png)
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

![SGEMM Performance Chart](chart.png)

<table>
<thead>
<tr>
<th>Number of nodes x number of cores per node</th>
<th>Base</th>
<th>Tuned</th>
<th>ScalaPack</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x8</td>
<td>200</td>
<td>400</td>
<td>600</td>
</tr>
<tr>
<td>4x8</td>
<td>400</td>
<td>600</td>
<td>800</td>
</tr>
<tr>
<td>8x8</td>
<td>600</td>
<td>800</td>
<td>1000</td>
</tr>
</tbody>
</table>

OSU / Rice / CSU
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

   - 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

   - (Manual) Tuner capabilities
   - Distributed CnC

   - Ranges, polyhedra, union of polyhedra
   - No explicit modeling of task/data placement and communications
Contributions

- (Macro-) Dataflow Programming
- Dynamic Single Assignment
- (Manual) Tuners
- Shared + Distributed

DFGL

- Slight variation of CnC
- Recognition of Polyhedral subsets, Automatic coarsening
- Hierarchy of concepts for sets

PIPES

- Compiler support for "graph" transformations (e.g., coarsening, coalescing)
- Language constructs for task/data placement, and communications
- Automatic generation of Intel CnC C++ tuners
Core PIPES Language Features

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

```c
// 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

```plaintext
// 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

```plaintext
1 (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

```plaintext
1 (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

```
1 (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

```
1  [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

```plaintext
[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

```
1  [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

```
1 [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

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

Parser:
- Levels of syntactic sugar
- Extracts IR
- Nodes marked as Affine/non-Affine

Two paths:
- Affine and non-Affine
- Affine path is ISL based, extracts polyhedral abstractions, performs transformations, generates code, computes buffer sizes with the Barvinok library

PIPES Input

PIPES Front-End

PIPES Core
- Non-Affine Module
- Affine Module

PIPES Pretty-Printer

PIPES Output
- CnC C++ with PIPES API

C++ Compiler

CnC Binary

- Generates CnC code structure
- Inserts code snippets from PIPES core
- Prints PIPES - CnC API functions and macros

- PIPES API simplifies and encapsulates CnC's API
- Passes PIPES block collections to kernel steps

PIPES Externals (User-provided black boxes)
- Kernel step functions
- Initializers (e.g. reading data)
- Finalizers (e.g. freeing buffers)
Automatic Transformations:

**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
Automatic Transformations:

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

\[-\mathbf{u}. \mathbf{n} - w \leq \Theta_R(\mathbf{x}_R) - \Theta_S(\mathbf{x}_S) \leq \mathbf{u}. \mathbf{n} + w\]

- High potential to minimize run-time overhead

```plaintext
1   ...
2   // Task prescriptions
3   env :: (mul:1..N,1..N,1..N);
4   env :: (add:1..N,1..N,1..N);
5   // Tiling by 100x100x100 =>
6   // 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

```
1 ...  
2 // Task prescriptions  
3 env :: (mul:1..N,1..N,1..N);  
4 env :: (add:1..N,1..N,1..N);  
5 // Task dataflow  
6 [A:i,k],[B:k,j] -> (mul:i,j,k) -> [C:i,j,k];  
7 [C:i,j,k],[D:i,j,k] -> (add:i,j,k) -> [D:i,j,k+1];  
8 coalesce ([mul:i,j,k],[add:i,j,k]);  
9 // result:  
10 // [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

PIPEDS_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

```
1: for each block collection b do
2:   rank_consumers = ∅
3:   for each producer relation p → b do
4:     for each consumer relation b → c do
5:       task_consumer = (p → b) ◦ (b → c)
6:       affinity_map = find_affinity_map (IMG(task_consumer))
7:       rank_consumers += IMG(affinity_map)
8:     end for
9:   end for
10:  codegen (b, rank_consumers, consumed_on)
11: 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

```
1: for each block collection b do
2:   rank_consumers = ∅
3:   for each producer relation p \rightarrow b do
4:     for each consumer relation b \rightarrow c do
5:       task_consumer = (p \rightarrow b) \circ (b \rightarrow c)
6:       affinity_map = find_affinity_map (IMG(task_consumer))
7:       rank_consumers += IMG(affinity_map)
8:     end for
9:   end for
10:  codegen (b, rank_consumers, consumed_on)
11: 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

```
1: for each block collection b do
2:   rank_consumers = Ø
3:   for each producer relation p → b do
4:     for each consumer relation b → c do
5:       task_consumer = (p → b) ◦ (b → c)
6:       affinity_map = find_affinity_map (IMG(task_consumer))
7:       rank_consumers += IMG(affinity_map)
8:     end for
9:   end for
10:  codegen (b, rank_consumers, consumed_on)
11: 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

1: for each block collection $b$ do
2:   rank_consumers = $\emptyset$
3:   for each producer relation $p \rightarrow b$ do
4:     for each consumer relation $b \rightarrow c$ do
5:       task_consumer = $(p \rightarrow b) \circ (b \rightarrow c)$
6:       affinity_map = find_affinity_map($IMG(task_consumer)$)
7:       rank_consumers += $IMG(affinity_map)$
8:     end for
9:   end for
10:  codegen ($b$, rank_consumers, consumed_on)
11: end for
if (kk0 >= 0 && N >= 1000 * kk0 + 1 && kk1 >= 0 && N >= 1000 * 
k1 + 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 * 
k1 + 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)

1: for each block collection $b$ do
2: rank_producers = $\emptyset$
3: for each consumer relation $b \rightarrow c$ do
4: for each producer relation $p \rightarrow b$ do
5: task_producer = $(b \rightarrow c)^{-1} \circ (p \rightarrow b)^{-1}$
6: affinity_map = find_affinity_map ($IMG(task_producer)$)
7: rank_producers += $IMG(affinity_map)$
8: end for
9: end for
10: codegen ($b$, rank_producers, produced_on)
11: 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)

1: for each block collection \( b \) do
2:   rank_producers = \( \emptyset \)
3:   for each consumer relation \( b \rightarrow c \) do
4:     for each producer relation \( p \rightarrow b \) do
5:       task_producer = \( (b \rightarrow c)^{-1} \circ (p \rightarrow b)^{-1} \)
6:       affinity_map = find_affinity_map (\( IMG(task_producer) \))
7:       rank_producers += IMG(affinity_map)
8:   end for
9: end for
10: codegen (b, rank_producers, produced_on)
11: 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)

```plaintext
1: for each block collection b do
2:    rank_producers = ∅
3:    for each consumer relation b → c do
4:      for each producer relation p → b do
5:        task_producer = (b → c)^{-1} ∘ (p → b)^{-1}
6:        affinity_map = find_affinity_map (IMG(task_producer))
7:        rank_producers += IMG(affinity_map)
8:      end for
9:    end for
10:   codegen (b, rank_producers, produced_on)
11:  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)

```plaintext
1: for each block collection b do
2:   rank_producers = ⌀
3:   for each consumer relation b → c do
4:     for each producer relation p → b do
5:       task_producer = (b → c)⁻¹ ∘ (p → b)⁻¹
6:       affinity_map = find_affinity_map (IMG(task_producer))
7:       rank_producers += IMG(affinity_map)
8:     end for
9:   end for
10:  codegen (b, rank_producers, produced_on)
11: 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

```plaintext
1: for each task collection t do
2:    deps = ∅
3:   for each consumer relation b → t do
4:      (t → b) = (b → t)^{-1}
5:     Convert input tuple dimensions of (t → b) to parameters
6:     deps += IMG(t → b)
7:   end for
8: codegen (t, deps, dependency_consumer)
9: 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

---

1: for each task collection t do
2:   deps = ∅
3: for each consumer relation $b \rightarrow t$ do
4:   $(t \rightarrow b) = (b \rightarrow t)^{-1}$
5: Convert input tuple dimensions of $(t \rightarrow b)$ to parameters
6:   deps += $IMG(t \rightarrow b)$
7: end for
8: codegen (t, deps, dependency_consumer)
9: 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)

```
1:   for each block collection b do
2:     for each consumer relation b → c do
3:       count = 0
4:       Convert input tuple dimensions of b → c to parameters
5:         for each task t in IMG(b → c) do
6:           count += CARD(t)
7:         end for
8:     end for
9:   codegen (b, count, maxlife)
10:  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)

```
1: for each block collection b do
2:   for each consumer relation b → c do
3:     count = 0
4:     Convert input tuple dimensions of b → c to parameters
5:     for each task t in IMG(b → c) do
6:       count += CARD(t)
7:     end for
8:   end for
9:  codegen (b, count, maxlife)
10: end for
```
## Experimental Setup

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

*Table: Experimental setup*
SSYR2K

The Symmetric rank-2 update computes the function:

```plaintext
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:

```c
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 $SGEMM(C,A,\text{trans}(B))$

1. $[C:i,j,k], [A:i,k], [B:j,k] \rightarrow (\text{GEMM}:i,j,k,0) \rightarrow [C:i,j,k+1];$
2. $[D:i,j,k], [B:i,k], [A:j,k] \rightarrow (\text{GEMM}:i,j,k,1) \rightarrow [D:i,j,k+1];$
3. $[C:i,j,N], [D:j,i,N] \rightarrow (\text{AddMat}:i,j) \rightarrow [\text{Res}:i,j];$

Figure: Parallel SYR2K

Transposed variant exploits:

\[
\text{trans}(SGEMM(\text{trans}(A),\text{trans}(B))) = SGEMM(A,B)
\]

1. $[C:i,j,k], [A:i,k], [B:j,k] \rightarrow (\text{GEMM}:i,j,k,0) \rightarrow [C:i,j,k+1];$
2. $[D:i,j,k], [B:j,k], [A:i,k] \rightarrow (\text{GEMM}:i,j,k,1) \rightarrow [D:i,j,k+1];$
3. $[C:i,j,N], [D:j,i,N] \rightarrow (\text{AddMat}:i,j) \rightarrow [\text{Res}:i,j];$

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
Results:

2MM

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

Performance Contribution of CnC Tuners

- a) Untuned
- b) Compute On Tuner (Affinity)
- c) (b)+Dependency Scheduler
- d) (c)+Get Count Tuner
- e) (c)+Consumed On Tuner (Push Model)
- f) (e)+Get Count Tuner
### PIPES Statistics

<table>
<thead>
<tr>
<th>Variant</th>
<th>Input Lines</th>
<th>Output Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEMM Cannon</td>
<td>40</td>
<td>1050</td>
</tr>
<tr>
<td>SGEMM Johnson</td>
<td>30</td>
<td>900</td>
</tr>
<tr>
<td>SSYR2K Parallel Cannon</td>
<td>55</td>
<td>1700</td>
</tr>
<tr>
<td>SSYR2K Transposed Cannon</td>
<td>45</td>
<td>1400</td>
</tr>
<tr>
<td>SSYR2K Parallel Johnson</td>
<td>40</td>
<td>1200</td>
</tr>
<tr>
<td>SSYR2K Transposed Johnson</td>
<td>40</td>
<td>1000</td>
</tr>
</tbody>
</table>

**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-performance
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