Polyhedral Network Aware Task Scheduling

Martin Kong

Brookhaven National Laboratory

October 14th, 2017
9th Annual Concurrent Collections Workshop
College Station, TX, USA

BROOKHAVEN NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

Introduction

- Context: Shared and distributed memory computers, or any hardware where the underlying network can be modeled
- We introduce different virtual network topologies that allow to model concepts such as direction and distance in order to produce a task schedule
- As an application, we choose the task isolation problem: how to schedule tasks which access shared resources
- Networks and hierarchies are pervasive in computer science, i.e. memory systems, computation graphs, programs
- Concrete instances of this problem are:
  - False sharing
  - Placing tasks close or nearby to fixed shared resources
  - Scheduling access of R/W resources (e.g. a file or freshly created data block)
  - Job scheduling and accessing a file system (explicit hierarchy) with fast and slow storage systems

Motivation

- Large scale applications require substantial domain, algorithmic and hardware/software knowledge
- Current and emerging technologies pose an enormous challenge in terms of performance portability and user productivity
- Data movement and long latencies are strong limiting factors for performance
- Knowledge of the underlying network topology, even in shared-memory machines, is essential to performance tuning
- Diverse background of users calls for abstractions that allow to easily switch between prescriptive and descriptive programming models and methods
- New abstractions are necessary in order to exploit and integrate network-aware compiler optimizations which:
  - Avoid communication
  - Perform efficient communication: minimize synchronization and data movement
  - Schedule tasks around data
  - Transfer domain knowledge to the compiler

A Look into the (near) Future

Feature	Titan	Summit
Application Performance	Baseline	5-10x Titan
Number of Nodes	18,688	~4,600
Node Performance	1.4 TF	> 40 TF
Feature	32 GB DDR3 + 6 GB GDDR5	512 GB DDR4 + HBM
NV Memory per Node	0	1600 GB
Total System Memory	710 TB	10 PB DDR4 + HBM + Non-Volatile
System Interconnect (Injection Bandwidth)	Gemini (6.4 GB/s)	Dual Rail EDR-IB (23 GB/s)
Interconnect Topology	3D Torus	Non-blocking Fat Tree
Procesor	1 AMD Opteron + 1 NVIDIA Kepler	2 IBM Power9 + 6 NVIDIA Volta
File System	32 PB, 1 TB/s, Lustre	250 PB, 2.5 TB/s GPFS
Peak Power	9 MW	15 MW

Context

- Previously considered data dependences and “performance dependences” (e.g. additional dependences that affect scheduling, directly or indirectly)
- We leverage polyhedral tools to model different network topologies
- Goal: composability of networks
- Polyhedral compilation frameworks leverage lexicographic minimization
- Optimization of Manhattan-distance type problems are not immediately modelable
- In this work, we provide abstractions for modeling different types of network topologies
- We follow an iterative optimization process

Contributions

CnC[1]

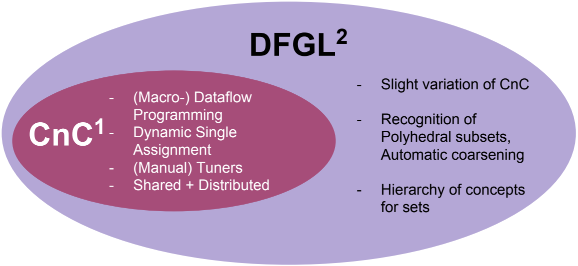
- (Macro-) Dataflow Programming
- Dynamic Single Assignment
- (Manual) Tuners
- Shared + Distributed

[1] Budimlic et. al, "Concurrent Collections", Scientific Programming, 2010

[2] A. Sbirlea, L.-N Pouchet and V. Sarkar, "DFGR: an intermediate graph representation for macro-dataflow programs", DFM, 2014

Kong, Pouchet, Sadayappan, Sarkar, "PIPES: A Language and Compiler for Task-Based Programming on Distributed-Memory Clusters", SC, 2016

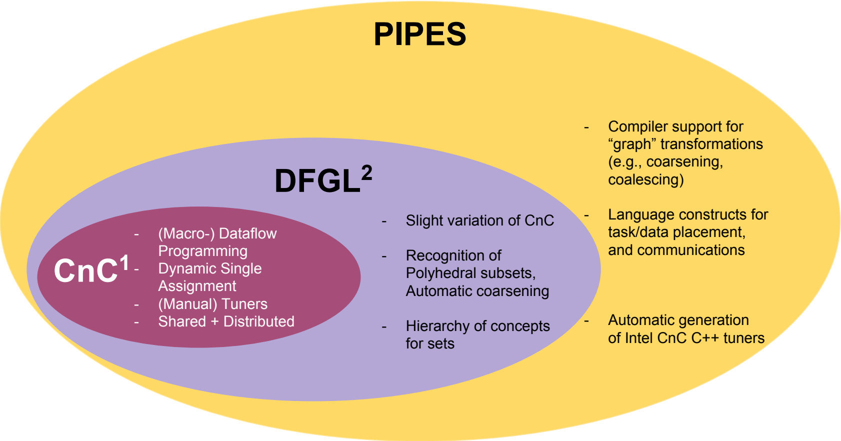
Contributions



[1] Budimlic et. al, "Concurrent Collections", Scientific Programming, 2010
[2] A. Sbirlea, L.-N Pouchet and V. Sarkar, "DFGR: an intermediate graph representation for macro-dataflow programs", DFM, 2014

Kong, Pouchet, Sadayappan, Sarkar, "PIPES: A Language and Compiler for Task-Based Programming on Distributed-Memory Clusters", SC, 2016

Contributions

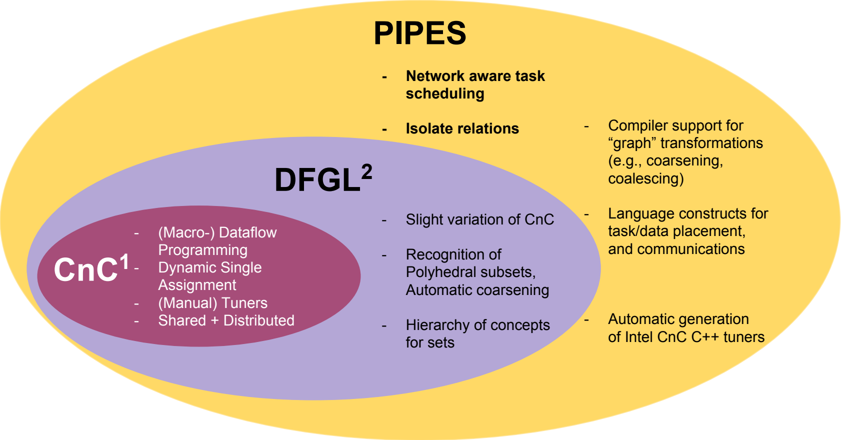


[1] Budimlic et. al, “Concurrent Collections”, Scientific Programming, 2010

[2] A. Sbirlea, L.-N Pouchet and V. Sarkar, “DFGR: an intermediate graph representation for macro-dataflow programs”, DFM, 2014

Kong, Pouchet, Sadayappan, Sarkar, “PIPES: A Language and Compiler for Task-Based Programming on Distributed-Memory Clusters”, SC, 2016

Contributions

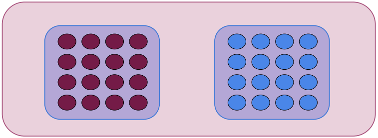


[1] Budimlic et. al, "Concurrent Collections", Scientific Programming, 2010

[2] A. Sbirlea, L.-N Pouchet and V. Sarkar, "DFGR: an intermediate graph representation for macro-dataflow programs", DFM, 2014

Kong, Pouchet, Sadayappan, Sarkar, "PIPES: A Language and Compiler for Task-Based Programming on Distributed-Memory Clusters", SC, 2016

PIPES Language Features



PIPES allows to declare tasks with ranges, polyhedra, union of polyhedra and arbitrary sets

- Language features are task-centric
- Virtual topologies
- Task placement
- Data placement
- Data communication (pull or push communication model)

MatMul in PIPES

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

Figure: PIPES Matrix Multiplication

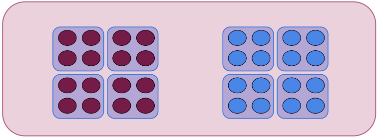
Language Construct Summary

Name	Syntax
Region	regname = { tuple : constraints } properties
Prescription Relation	env :: (task : regname)
Data-Flow Relations	[input_instances] -> (task_instance) -> [output_instance]
Virtual Topology	Topology topo_name = { sizes=[parameter_list] }
Affinity Mapping	(task : tuple) @ [[topo : tuple]]
Communication	[item: tuple] @ (task1 : tuple) => (task2 : tuple)
Scheduling	(task1 : tuple) -> (task2 : tuple)
	(task1 : tuple) ∼> (task2 : tuple)

Table: PIPES Language Constructs

Virtual Topologies and Task Mapping

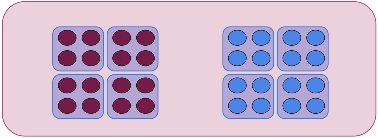
- Virtual topologies (VTs) represent the logical underlying computer grid/cluster
- Each element in the set is a processor
- Requires a logical-to-physical mapping



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

Virtual Topologies and Task Mapping

- 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) @ Topo2d(point);
```

Overall Approach

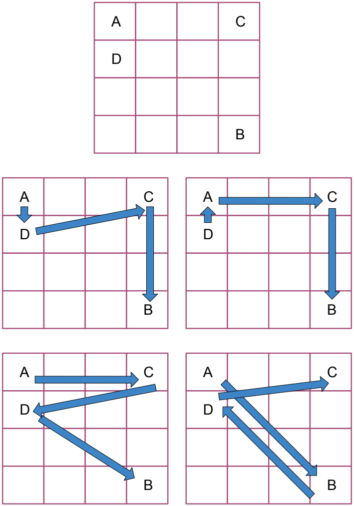
- Network topology abstractions
- Map task instances to virtual topology via Affinity Tasks Mappings (ATM)
- Introduce **isolate relations** and **isolate dependences**
- Compute in closed for the possible task interleavings
- Filter / prune unwanted tasks orderings from the closed form
- For each point in the closed form, iteratively compute its cost
- Apply lexicographic minimization to determine the best execution order

Network Topologies

- PIPES uses an abstraction that allow to pin task instances to processing elements / cores; main purpose is to determine if tasks execute on the same location; then generate Intel CnC++ tuners
- In this work: Implemented two different topologies: mesh and fat tree
- Pending to implement: torus / meshes + wraparound, hypercubes, hierarchies
- Each network type requires different set of abstractions
- Abstractions allow modeling concepts such as dimension, distance or direction
- In the future, want to pursue composing two or more topology types, so as to faithfully represent upcoming HPC and Data Analytics clusters

Driving Example

- Consider a 4x4 mesh and 4 tasks (A,B,C,D)
- What if A has to start the computation?
- What if any task can start the computation?
- What if wrap-around communication is allowed?
- In general, avoiding ping-pong-ing around

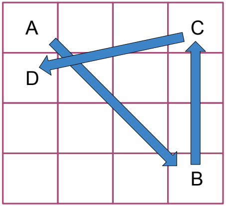


Network Topologies: N-dimensional Meshes

Key: idea: translate task tuple to network coordinates, then compute distance between network coordinates.

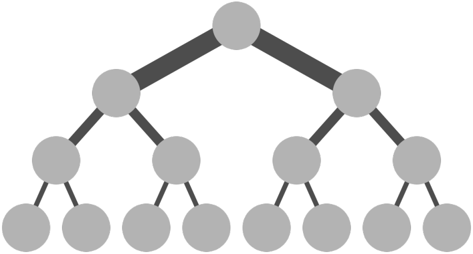
Example:

1. Compute task topology coordinates:
   A=G[0,0]; B=G[3,3];
   C=G[0,3]; D=G[1,0]
2. Compute task-to-task topology directions:
   Dir(A,B) = [3,3]; Dir(B,C) = [0,-3];
   Dir(C,D) = [-3,1]
3. Compute "positive directions":
   +Dir(Dir(A,B)) = [3,3];
   +Dir(Dir(B,C)) = [0,3];
   +Dir(Dir(C,D)) = [3,1];
4. Compute task-to-task topology distance:
   Dist(A,B) = MultiplexAddMap([3,3]) = 6
   Dist(B,C) = MultiplexAddMap([0,3]) = 3
   Dist(C,D) = MultiplexAddMap([3,1]) = 4



Network Topologies: Fat Trees

- Distance metrics in fat tree type of network are non-linear
- Coordinate space is one dimensional
- Task tuples must be "flattened" to represent processors in the grid
- Restricted to fixed (non-parametric) network sizes
- Approach: explicitly construct a distance map from a pair of processors to a fixed non-parameteric value, i.e. $[[P_i]->[P_j]]->[distance]$
- Does not require the **coordinate to direction** map nor the **positive direction map**
- Observe the recursive nature



	P0	P1	P2	P3	P4	P5	P6	P7
P0	0	1	2	2	4	4	4	4
P1	1	0	2	2	4	4	4	4
P2	2	2	0	1	4	4	4	4
P3	2	2	1	0	4	4	4	4
P4	4	4	4	4	0	1	2	2
P5	4	4	4	4	1	0	2	2
P6	4	4	4	4	2	2	0	1
P7	4	4	4	4	2	2	1	0

Network Topology Abstractions

- Affinity maps (task tuple to network coordinate maps)
- Coordinate to direction maps (each dimension can be +/-)
- Direction unification map: consider different routes along each dimension and direction (several disjunctions), e.g. in meshes with wraparound
- Direction to distance maps: make directions positive
- Cummulative distance maps: “Multiplex Add Map”

Building the Closed Form of the Exploration Space

- Compact and closed form for encoding task orderings at the coarse level
- Assign to each task a fixed integer
- Start with the T^T potential task orderings (T : number of **lexical** tasks), fixed and known at compile time
- The exploration space consist of the set of points in a T-dimensional tuple space.
- Add bounding constraints: $\vec{t} = (t_1, t_2, ..., t_T)$, $\forall\ t_i \in [1..T]$
- Add constraints to remove meaningless points, e.g. (1,1,1) which represents the same task being executed: $\sum t_i = T \times (T+1)/2$
- Add constraints to enforce data dependences, i.e. if task 1 is a dependence of task 2 then: $t_i = 1$ and $t_j = 2$, for some $j > i$
- Each point in the set represents a full execution path (e.g. $3 \rightarrow 1 \rightarrow 4 \rightarrow 2$)

Pruning the Scheduling Space

We consider three types of pruning constraints that are added to the complete space, e.g. $(t_1, t_2, ..., t_T), t_i \in [1..T]$

- Data dependence edges: "A(1) must to execute before B(2)"
- Scheduling edges (for performance): "We want A(1) to execute before B(2)"
- **Isolation edges**: "We want either A(1) before B(2) or B(2) before A(1)"

Several disjunctions might be required to model the constraints, in particular for the **isolation edges**

A few more considerations:

- In theory, we could still have meaningless points, e.g. (1,1,4,4) = 10
- In practice, these would be mostly naturally pruned by the above constraints
- To be safe, we do a manual check before the next stage to guarantee that all entries in a vector are distinct

Iteratively Computing Path Costs

space $\leftarrow$ compute_closed_form (T,program)
for each $\vec{t} = (t_1, t_2, ..., t_N) \in space$ **do**
  path($\vec{t}$) $\leftarrow$ 0
  for each $(t_i, t_{i+1}) \in (t_1, t_2, ..., t_N)$ **do**
    Fetch task domains $T(t_i)$ and (T_{i+1})
    Fetch task affinity maps: $AM(t_i)$ and $AM(T_{i+1})$
    Build synchronization map: **sync_map** $\leftarrow T(t_i) \rightarrow T(t_{i+1})$
    Intersect dependences $T(t_i) \Rightarrow T(t_{i+1})$ with **sync_map**
    Compute processor synchronization map (PSM): PSM $\leftarrow AM(t_i)^{-1} \circ sync \circ AM(T_{i+1})$
    Compute "processor coordinate difference" (PCD) with ISL deltas_map (PSM)
    Apply network specific coordinate-to-distance map to PCD
    edge(i) $\leftarrow$ quasi_polynomial_sum(PSM)
    path($\vec{t}$) $\leftarrow$ path($\vec{t}$) + edge(i)
  end for
  M $\leftarrow$ M $\cup$ path($\vec{t}$)
end for
result $\leftarrow$ lexmin(M)

Applications of this Technique

- Minimize synchronization latency among tasks
- Pre-optimization pass that affects the overall program prior to applying high-level loop transformations that optimize for locality, akin to code motion in for loops
- Allows to establish order among tasks that should be executed in a non-concurrent fashion:
  1. User provides **isolate relations**, e.g. "$A \sim\parallel B$", another class of dependence that will be enforce semantic orderings, both at the compiler and runtime level
  2. Compiler "decides the direction" based on network distance/latency
  3. Isolate relations are then promoted to "isolate dependences" and fed to some other scheduler

Applications of this Technique

- This technique has the potential to reduce the runtime scheduling overhead by substantially narrowing down the scheduling options
- Autotuning: coupled with auto-generated task mappings, allows to determine suitable mappings and task schedules
- Applicable to several task parallel and data-flow runtimes (CnC!!)

Restrictions of the Approach

- Very computational expensive, even in non-parametric cases
- Limited to ~ 10 tasks (millions of possible interleavings) and taking ~ 10 min
- Disallow edges that induce loops in the graph
- Some networks cannot be modeled with affine parametrics constraints, resort to use large fixed values and bound network parameters by the context

Experimental Setup

- OS: Mac Sierra
- 3.5 GHz Intel Core i7
- Memory:16 GB 2133 MHz
- Compiler: Clang++ Apple LLVM version 8.1.0 (clang-802.0.42)
- Barvinok 40
- ISL 18.0

Will show some preliminary compilation results for 2-D meshes (with fixed and parametric task domains) and fat-trees.

Mesh Results: Fixed

Test	Tasks	Deps	Factorial	Legal	Semi-legal	Complete	Time (sec)
1	2	1	2	1	2	4	0.021
2	2	0	2	2	2	4	0.059
3	2	0	2	2	2	4	0.101
4	3	2	6	2	4	27	0.153
5	5	4	120	2	130	3125	0.205
6	2	1	2	2	2	4	0.027
7	3	2	6	2	4	27	0.161
8	3	3	6	4	4	27	0.069
9	3	3	6	4	4	27	0.094

Space exploration size (T : number of lexical tasks):

- Factorial: T!
- Legal: Final exploration space
- Semi-Legal: Exploration space before adding dependence edges
- Complete: T^T exploration space

Mesh Results: Parametric

Test	Tasks	Deps	Factorial	Legal	Semi-legal	Complete	Time (sec)
01	2	0	2	2	2	4	0.056
02	3	2	6	2	7	27	
03	3	1	6	3	7	27	0.037
04	4	1	24	17	44	256	0.1
05	5	1	120	120	381	3,125	
06	5	4	120	4	381	3,125	
07	6	5	720	12	4,332	46,656	0.15
08	7	5	5,040	84	60,691	823,543	0.96
09	9	7	362,880	648	19,610,233	387,420,489	233
10	9	8	362,880	1	19,610,233	387,420,489	28.8
11	10	9	3.63E+06	2	432457640	10,000,000,000	317.2

Space exploration size (T : number of lexical tasks):

- Factorial: T!
- Legal: Final exploration space
- Semi-Legal: Exploration space before adding dependence edges
- Complete: T^T exploration space

Fat Tree Results

Test	Fat Tree Max Size	Tasks	Deps	Factorial	Legal	Semi-legal	Complete	Time (min + sec)
1	16	2	1	2	2	2	4	0m0.077s
2	32	2	1	2	2	2	4	0m0.167s
3	64	2	1	2	2	2	4	0m0.495s
4	128	2	1	2	2	2	4	0m1.583s
5	256	2	1	2	2	2	4	0m5.556s
6	512	2	1	2	2	2	4	0m21.163s
7	1024	2	1	2	2	2	4	1m21.719s
8	2048	2	1	2	2	2	4	5m31.795s
9	256	5	3	120	12	130	3125	2m5.986s
10	256	5	4	120	34	130	3125	5m58.886s
11	256	5	6	120	130	130	3125	22m57.720s

Space exploration size (T : number of lexical tasks):

- Factorial: T!
- Legal: Final exploration space
- Semi-Legal: Exploration space before adding dependence edges
- Complete: T^T exploration space

Future Work

- Complete implementation of torus networks, hypercubes and explicit hierarchies
- Enable composability of network topology abstractions
- Integrate resource features into topology abstraction:
  - To model **super nodes** that have access near to accelerators or to memory nodes
  - Specialized resource types e.g. streaming nodes, compute intensive nodes, memory nodes, storage nodes, etc
- Potential direction: focus on file access scheduling (e.g. in HPF5) by using the data file and block structure to construct the program graph
- Complete integration to PIPES compiler
- Leverage the newly introduced constructs to efficiently implement and perform communication patterns such as:
  - All to all communication
  - Multicast and broadcast
  - Nearest neighbor type of communication
- Improve the scalability of the technique: Perform cuts on the graph, search for "bridge tasks" and "articulation tasks"

Das Ende

Thanks for listening

Questions?