Easy and Efficient Graph Analysis: A DSL-based Approach

Sungpack Hong and Kunle Olukotun

Pervasive Parallelism Laboratory
Stanford University
Graph Analysis

- **Graph**
  - Fundamental data representation
  - Captures random relationship between data entities
  - You learned about it in CS 101

- **Why graph once again?**
  - New applications (in lucrative markets) use graph analysis—social networks, computational biology, ...
    - e.g. Analyze molecular interaction graph in your body cells to identify key proteins
  - Requires significant processing power
    - Underlying graph size is large and growing
    - Some algorithms are expensive, i.e. $O(n^2)$ or more
    - Classic ILP, Vector Units, FLOPs does not help much

  ➔ Let’s use parallelism
Parallel graph analysis

- **Opportunities**
  - Plenty of inherent (data-) parallelism in large graph instances
  - 100+ years of studies in graph theory
  - Parallel machines are now and everywhere (Multi-core CPU and GPU)

- **Challenges**
  - Hard to get correct implementation
  - Performance depends on implementation (even with the same algorithm)
  - The best implementation differs from machine to machine
  - Algorithms need to be customized
What’s wrong with libraries?

- There are 30+ graph libraries/packages
- Issues in fixed library implementation
  - Parallelism?
  - Portability?
  - New/customized algorithm?
  - What if someone finds a better implementation for the un-customized algorithm?
Our approach: DSL for graph analysis

1. Identify key components in graph algorithms as define them language constructs.
2. Find (the best / a good) implementation of those constructs.
3. Let the compiler translate high-level algorithm written in DSL into a high-performing low-level implementation.
   - Possibly, apply high-level optimization on the way
Approaches

- DSL design
- Implementation of language constructs
  - BFS for GPU
  - BFS for CPU
- Compiler development
Language Design

- **Domain property**
  - The graphs are *sparse, small-world, scale-free*
    - Graph is not mesh-like!
  - Graph modification is less frequent than graph analysis

- **Language Design: an inductive process**
  - Examine existing algorithms ➔ Extract language constructs
  - Check if these algorithms can be naturally expressed with the language
  - Check if the compiler can figure out inherent parallelism from the description.

*The DSL is named as Green-Marl which means graph language (그림 말) in Korean.*
A Glimpse of DSL Syntax

- Example: Betweenness Centrality
  - A measure that tells how center a node is located in the graph
  - Frequently used in social network analysis
  - Computationally expensive: $O(NM)$
A Glimpse of DSL Syntax

Original Algorithm

Algorithm 1: Betweenness centrality in unweighted graphs

$$C_B[v] \leftarrow 0, \forall v \in V$$

for \( s \in V \) do

Compute sigma from parents

Compute delta from children

Procedure comp_BC(G: Graph, BC: Node_Property<Float>(G))

```java
G.BC = 0; // Initialize
Foreach (s: G.Nodes) {
    // temporary values per Node
    Node_Property<Float>(G) sigma;
    Node_Property<Float>(G) delta;
    G.sigma = 0; // Initialize
    G.delta = 0;
    s.sigma = 1;
    // BFS order iteration from s
    InBFS(v: G.Nodes From s) {
        v.sigma = // Summing over BFS parents
        Sum (w:v.UpNbrs) {w.sigma};
    }
    // Reverse-BFS order iteration to s
    InRBFS(v: G.Nodes To s)(v != s) {
        v.delta = // Summing over BFS children
        Sum (w:v.DownNbrs) {
            v.sigma / w.sigma * (1+ w.delta) };
        v.BC += v.delta @ s; // accumulate BC
    }
}
```
Language Philosophy

- Goal is not to magically parallelize your sequential graph algorithm
  - Would you believe it, if I claim so?
  - People have devoted their entire career in developing parallel graph algorithms
- Instead, it allows you to express your algorithm (sequential or parallel) in a natural way
- The compiler grabs out the inherent parallelism in the algorithm and exploit it in the implementation
  - e.g. Betweenness Centrality is not designed for parallel execution
Consistency Model

- We are targeting different architectures: CPU, GPU, (Cluster)
- The language, thus, assumes the most relaxed form of consistency

```java
Foreach (t: G.nodes) {
    Int z = Sum (v: t.Nbrs) {v.Val};
    t.Val = z;
}
```

- Conceptually, `Foreach` is instantiated all at the same time.
- There is no guarantee update to `Val` will be visible (or not) to other instances, until the end of foreach
- It’s not even Total Store Ordering

- Enforcing order out of chaos
  - High-level operations are atomic (e.g. add to a set)
  - Reduction and Deferred assignments
Consistency Model

- Deferred assignment

```plaintext
Foreach (t: G.Nodes) {
    Int z = Sum (v: t.Nbrs) {v.Val};
    t.Val <= z @ t;
}
```

- Reduction assignment

```plaintext
Foreach (v: G.Nodes) {
}
```

If the graph is undirected, we can exchange the iteration order to remove reduction.

Reduction by minimum is resolved at the end of foreach iteration bound by t.

Write to `Val` happens at the end of iteration bound by t.

CPU: comp & swap
GPU: atomicMin
Cluster: MapReduce
I need Sequential Consistency!

- Is this what you want?

```java
Foreach (t: G.nodes) {
    Atomic {
        Int z = Sum (v: t.Nbrs) {v.Val};
        t.Val = z;
    } @ t
}
```

- Your algorithm is not deterministic, you know
- We may add it to the language, though
  - Coloring like Listz [Big setup overhead]
  - Grab a lock of neighbors
  - Performance is not guaranteed; due to the graph shape (i.e. not mesh)
Reduction Assignment vs. Reduction Operator

- **Reduction Assignment (spread-out)**

```
Int z = 0;
Foreach (n: G.Nodes) {
    If (n.color == 0) { z += n.val @ n; }
    Else {
        Foreach (t: n.Nbrs)(t.color == 1) { z += t.val @ n; }
    }
}
```

- **Reduction Operator (in-place)**

```
Int z = 0;
Foreach (n: G.Nodes) {
    z = Sum (t: n.Nbrs)(t.color==0) {t.val};
}
```
A Few More on Syntax

- Nodes(Edges) are bound to a graph

```plaintext
Graph G1, G2;
Node(G1) t1;
Node(G2) t2;
t1 = t2;  // Type Error!
```

- Fields can be defined dynamically and passed as arguments

```plaintext
Graph G;
While (...) {
    Node_Property<Int>(G) cap;
    ...
    Foo(G, cap);
} // cap has static scope
```

Call common routines with different fields.

```plaintext
Procedure Foo
  (G:Graph, d:Node_Property<Int>(G))
{
    // ...
}
```
A Few More on Syntax

- **Sets**
  - Operation to a set is atomic: Add/Remove/IsIn
  - Set: bound to a graph

```plaintext
NodeSet(G) NSet;
EdgeSet(G) ESet;
NbrSet(G) NBSet;
NbrEdgeSet(G) NBESet;

Foreach (t: G.Nodes)
  Foreach (n: t.Nbrs)
    If (n.value > THRESHOLD) {
      t->NBSet.Add(n);
    }
```

A Few More on Syntax

- **Static Scope**
  - Variable name shadowing is not allowed.

```plaintext
Foreach (t: G.Nodes) {
    Int k;
    Foreach (n: t.Nbrs) {
        Int t; // Error;
    }
    Int n; // Okay;
}
```
Some Rules to be Enforced

- Cannot write to an iterator

```java
Node(G) n;
Foreach (t: G.Nodes) {
    n = t; // Okay
    t = n; // Error
}
```

- Cannot write to a property reference

```java
N_P<Int>(G) val;
N_P<Int>(G) cap;
Node(G) n;
n.val = n.cap // Okay;
G.val = G.cap // Okay;
cap = val; // Error;
```
Some Rules to be Enforced

- Reduce (Defer) Assignment should be bound once and only once.

```java
Int z = 0;
Foreach(t: G.Nodes) {
    z += t.val @ t;
    Foreach (n: t. Nbrs) {
        z += n.val @ t; // Okay
        z += n.val @ n; // Error
        z min= n.val @ t; // Error
    }
    z = 3; // Error
}
Z += 3; // Error
```
Parallelization

Assumption

- Graph is large
- Otherwise uninteresting.
- One operation is enough to consume all the cache & memory bandwidth

Strategy

- CPU: Parallelize inner-most graph-wide iteration
- GPU: two-level parallelization: sub-warp + thread

```java
Foreach (t: G.Nodes) {
  ...
  Foreach (n: G.Nodes) {
    ...
      Foreach (r: n.Nbrs) {
        }
      }
  }
}
```
Optimization after Parallel Region Decision

```
Foreach (s: G.Nodes) {
    Foreach (t: G.Nodes) {
        t.val += ... @ s;
    }
}
```

Reduction can be implemented with normal write

```
Foreach (s: G.Nodes) {
    N_P<Int> temp;
    Foreach (t: G.Nodes) {
        t.temp += ... @ t;
    }
    s.val += s.temp @ s;
}
```

No need to create (and delete) temp, O(N) times. (Move temp-define out of s-loop)
Language Implementation

- **Breadth-First Search (BFS)**
  - An systematical way of traversing a graph
  - Enforces a natural (partial) ordering of the graph
  - Serves as a building block for other algorithms
    (Connected components, Betweenness centrality, Max flow computation...)
  - Many papers about efficient BFS implementation
    (Multi-Core CPU, GPU, Cell, Cray XMT, Cluster) ...
BFS on GPU

- Potentials of GPU in graph analysis
  - Large memory bandwidth (but with limited capacity)
    - + Latency hiding scheme
  - Massively parallel hardware
- Previous implementation [Harish and Narayanan 2007]
  - Level synchronous, frontier-expansion method.
  - PRAM-style; each thread processes a node.
  - Problem:
    - Performance dropped heavily when applied to scale-free graphs (i.e. skewed degree distribution)
BFS on GPU

- What causes this?
  - The trait of GPU architecture → Threads in a warp are executed in a synchronous way
  - Skewed degree distribution → Intra-warp workload imbalance

- Our implementation
  [PPOPP 2011]
  - Work assignment
    → per a subset of warp
    → Trade off under-utilization and workload imbalance

A unit of work per each thread
A unit of work per each warp

测\[\text{Previous}\]

Measured on GTX275 (Tesla GPU)
BFS on Multi-core CPU

- Level-synchronous Parallel BFS

```
Algorithm 1 Level Synchronous Parallel BFS
1: procedure BFS(r:Node)
2:   V = C = Ø; N = {r}
3:   r.lev = level = 0
4:   repeat
5:     C = N ; N= Ø
6:     for Node c ∈ C do
7:       for Node n ∈ Nbr(c) do
8:         if n ∉ V then
9:           N = N ∪ {n}; V = V ∪ {n}
10:          n.lev = level + 1
11:     end
12:   until N = Ø
```

- Previous Implementation [Agarwal et al 2010]
  - Adopted a few techniques: prefetch, bitmap (Visited), non-blocking queue (Next/Curr Set)
  - Non-blocking queue: sophisticated implementation
    - Reduce synchronization and cache-cache coherence traffic.
    - Not much implementation details revealed in the paper.
Observations

- You don’t need a queue. You need a set.
- Cache traffic due to the queue is thus artificial.
- Performance is more governed by memory traffic (capacity miss) rather than coherence traffic.

Our approach [under submission]

- Implement Curr/Next set as a (single) byte-array.
  - Visited set is still a bitmap
- Cons
  - (Iteration over set) == (Read the whole byte array)
- Pros
  - No synchronization when writing
  - Sequential read when iterating

BFS on Multi-core CPU

```c
for Node c ∈ C do
```

```c
for (i=0;i<N;i++) {
    if (C[i] == curr) {
        ...
    }
}
```

Turns out to be okay, due to small-world property
BFS on Multi-core CPU

- Small world property?
  - A.k.a. six-degrees of separation
  - Diameter (maximum hop count between any two nodes) is small even with large graphs
    ➔ (# Nodes) in each BFS level grows, exponentially

LV0 LV1 LV2 LV3 LV4 LV5 LV6

Most execution time spent in these levels

O(N) nodes belong to a few levels in the middle

Time (ms)

LV0
LV1,2
LV3,4,5
LV6,7
Results

- 1.2x ~ 1.5x performance improvement
- Performance gap **widens** as graph size grows
- (+ Our algorithm is easier to implement)

Measured on Nehalem-EP CPU (2 socket x 4core x 2 HyperThread)
*For Fermi GPU, L1 has been disabled since it affected the performance negatively.

16 and 32 million nodes
avg degree = 8

Architectural Effects

2 socket * 4 core * 2 hyper-threading

Memory Bandwidth

2 socket * 4 core
DSL Compiler

- Currently under development
- Goal:
  - Maps language constructs with their best impl.
  - Source-to-Source translation.

[Diagram showing the process of DSL Compiler with a flow from DSL Description to DSL Compiler, then to Rewrite, Graph Analysis Routines, Parallel C++, CUDA, Link, and Your Complex Software']
Interfacing with user-world

- Translate entry function(s)
  - Arguments translation
  - Int → int32_t, Double → double, Set → Array, ...
  - Node/Edge/Graph → Library data type (node_t, edge_t, graph_t, ...)
  - Entry function should be called in a single-thread context (+ Whole GPU is available)

- Adopting user-defined functions, data types.
  - Like ASM in the C/C++
  - Simple text transformation
  - Bypass type-checking

```c
Procedure(G:Graph, val: N_P<Int>(G), z : $Utype)
{
    Foreach (t: G.Nodes)
    t.z = $UserFun (t.val, z);
}
```
Using other graph library

- Want to use other graph library?

Graph library may be replaced with other implementation (with small modification). However, the new graph library should allow parallel access at least.
Compiler is still under development
Result: Compiler Output

- **Sanity check**
  - Manual implementation of Betweenness Centrality (i.e. what the compiler should emit out.)
  - Showed $\sim 2x$ improvement
    - over a publicly available parallel implementation (8-core CPU)
    - Gain comes from using a better BFS scheme
Issues with Delite Implementation

- Syntax
  - dynamic property declaration
  - @ syntax

  ```
  N_P<Float>(G) X;
  G.X = 0;
  Foreach (t: G.Nodes) {
    If (t.flag) {
      Foreach (n: t.Nbrs) {
        n.X += t.cost @ t;
      }
    }
    t.X += t.cost @ t;
  }
  }
  ```

  ```
  Val X = N_P[Float](X, G);
  X = 0;
  Foreach (t <- Nodes(G)) {
    If (flag(t)) {
      Foreach (n <- Nbrs(t)) {
        X(n) += (cost(t), t);
      }
    }
    X(t) += (cost(t), t);
  }
  ```

- Rule Enforcing
  - Reduction rules
  - “UpNbrs” is only meaningful inside BFS.
Issues with Delite Implementation

- Transformation
  - Patterns that are far from each other
  - Lack of Symbol table
- Parallel Execution Strategy
- Code generation
  - CUDA
  - BFS Pattern

```c
{ ... 
  InBFS (v: G.Nodes From s) {
  }
  ... // some sentences
  If (...) {
    InRBFS (v: G.Nodes To s) {...}
  }
}
```
Issues with Delite Implementation

Lex → Parse → Type Check → Transformation → Optimization → Code Generation

- Syntax has to be modified
- Type check is free.
- Any other rules I make, I have to enforce them by myself
- Transformation should be described as pattern-matching.
- Optimization and Parallelization are independent
- Custom code generation patterns (e.g. BFS)

How many Delite-Ops do I use?
Distributed Graph Processing (Future Works)

- **Fundamental Issue**
  - Graph: random, small world, scale-free
    - Far from planar
    - Impossible to find a *good* partition
    - Surface to volume ratio is high
    - Communication overhead dominates

- **Pregel**
  - Google’s *framework* for distributed graph processing
  - Conceptually similar to MapReduce
    - Let’s just live with latency. Concentrate on bandwidth.
    - Bulk-Synchronous Consistency
    - A framework is provided - the user fills in custom computation.
    - However, the user function writing is not very intuitive.
Distributed Graph Processing (Future Works)

- PageRank Example

```java
class PageRankVertex
    : public Vertex<double, void, double>
    {
    public:
    virtual void Compute(MessageIterator* msgs) {
        if (superstep() >= 1) {
            double sum = 0;
            for (; !msgs->Done(); msgs->Next())
                sum += msgs->Value();
            *MutableValue() = 0.15 / NumVertices() + 0.85 * sum;
        }
        if (superstep() < 30) {
            const int64 n = GetOutEdgeIterator().size();
            SendMessageToAllNeighbors(GetValue() / n);
        } else {
            VoteToHalt();
        }
    }
    }

Procedure PageRank(G: Graph, e:Float, d:Float,
    p_rank: Node_Property<Float>(G)) {

    // Initialize
    Int N = G.NumNodes;
    G.p_rank = 1 / (Float) G.NumNodes;
    Float diff;
    Do {
        diff = 0;
        Foreach (t: G.Nodes) {
            Float new_val = (1-d) / N + d * Sum (w: t.InNbrs) {w.p_rank / w.NumOutNbrs} ;
            diff += |val - t.p_rank | @ t; // reduction assignment
            t.p_rank <= val @ t; // deferred assignment
        }
    } While (diff > e); // Iterate until converge
```

Can we find automatic translations?
Summary

- **DSL-based approach**
  - Productivity: Enables elegant algorithm description
  - Performance: Maps (best/good) parallel implementation
  - Portability: Generates CPU and GPU version
  - Flexibility: Language constructs are more than a library

- **Current Status**
  - A draft of language specification
  - Studies on BFS implementation
  - Prototype compiler on the way
Questions?

“Programs must be written for people to read, and only incidentally for machines to execute.”

– Abelson and Sussman
No more slides