Spatial Hardware Implementation for Sparse Graph Algorithms in GraphStep

Michael deLorimier, Nachiket Kapre, Nikil Mehta and André DeHon

How do we develop programs that are easy to express, easy to reason about, and able to achieve high performance on massively parallel machines? To address this problem, we introduce GraphStep, a domain-specific compute model that captures algorithms that act on static, irregular, sparse graphs. In GraphStep, algorithms are expressed directly without requiring the programmer to explicitly manage parallel synchronization, operation ordering, placement, or scheduling details. Problems in the sparse-graph domain are usually highly concurrent and communicate along graph edges. Exposing concurrency and communication structure allows scheduling of parallel operations and management of communication that is necessary for performance on a spatial computer. We study the performance of a semantic-network application, a shortest-path application and a max-flow/min-cut application. We introduce a language syntax for GraphStep applications. The total speedup over sequential versions of the applications studied is up to 4 orders of magnitude. The benefit of spatially-aware graph optimizations (e.g. placement and route scheduling) for the applications studied is up to a 30 times speedup.

Categories and Subject Descriptors: C.m [**Computer Systems Organization**]: Miscellaneous

General Terms: Languages, Algorithms, Performance

Additional Key Words and Phrases: Spatial Computing, Compute Model, Parallel Programming, Graph Algorithm, GraphStep

1. INTRODUCTION

Managing spatial locality is essential to extracting high performance from modern and future integrated circuits. Technology scaling is giving us more transistors, higher cross-chip communication latency relative to operation latency, and fewer cross chip wires relative to transistors. The first effect means we have more parallelism to exploit. The second two mean that communication optimizations are primary and are essential concerns that must be addressed to exploit the potential parallelism. Communication latency can dominate the critical path of the computation and interconnect throughput can be the performance bottleneck. By carefully selecting the location of operators and data in space, we can exploit parallelism effectively by minimizing signal latency and message traffic volume.

We introduce the GraphStep compute model [deLorimier et al. 2006]. The set of applications which GraphStep captures are those that are centered on large, static, sparse graphs. Typically the application iterates over steps where operations are performed on graph nodes, data is sent along edges, and there is a global broadcast to and reduce from nodes. We draw applications from domains such as semantic networks, CAD optimization,

numerical computations, and physical simulations. The applications we test are queries on ConceptNet, a semantic network, circuit retiming, which uses a shortest-path algorithm, and the max-flow/min-cut kernel for vision tasks.

By using the domain-specific GraphStep model we can map high-level, machine independent programs to spatially optimized implementations. In this domain, the graph structure captures the communication and computation structure that allows us to optimize for spatial locality. The domain-specific model abstracts out race conditions, synchronization details, operation and data placement, and operation scheduling.

We model the performance of GraphStep applications mapped to FPGA logic. FPGA hardware provides a highly parallel, spatial computing platform with the flexibility to support high communication bandwidth, high memory bandwidth, and low synchronization overhead. The logic architecture placed on the FPGAs is a collection of processing elements (PEs) interconnected with a Fat-Tree [Leiserson 1985] (Figure 6). Each PE has its own memory and compute logic.

The contributions of this work include:

1. We illustrate a concrete programming language for GraphStep (Figure 2) and give its formal semantics (Appendix A).
2. We quantify the benefit of a spatial implementation compared to a sequential implementation (Section 6).
3. We quantify the benefit of spatially-aware optimizations enabled by the GraphStep model, which are placement for locality, graph node decomposition, and static computation and communication scheduling (Section 5).

In Section 2 we explain the GraphStep model and compare it to other parallel compute models. Section 3 gives example GraphStep applications. Section 4 describes the hardware implementation. Section 5 describes and evaluates the optimizations performed on our example applications. Section 6 compares our example's performance in the GraphStep model to equivalent algorithms implemented sequentially. Section 7 concludes. Appendix A summarizes the formal semantics for GraphStep.

2. GRAPHSTEP MODEL

GraphStep is designed to express algorithms that work on sparse graphs. The computation structure follows the graph structure, so changes made to node state propagate changes along edges to neighboring nodes. This parallel activity is sequenced into steps, with one set of synchronous node updates and propagations per step. In general, a subset of nodes are updated in each step. A subset of the updated nodes then propagate changes to their neighbors. A sequential process initiates and controls parallel activity on the graph. It broadcasts to nodes and receives global reductions from nodes.

An operation on a node or edge generates messages which trigger operations on neighboring nodes and edges. The static graph structure is a directed multi-graph, so nodes send messages to their outgoing edges, and edges send messages to their destination nodes. The atomic action of an operation is to (1) input incoming message state along with the object state, (2) update object state, and (3) output new messages. The invocation of an operation is called an *operation firing*.

To match the common iterative structure of graph algorithms, this message passing activity is divided into *graph steps*. A graph step consists of three phases:

1. **reduce** – Each node performs a reduction on the incoming messages received along its

input edges. The reduction should be associative and commutative.
2. **update** and **send** – Each node with a reduction result updates its state and outputs messages to its output edges. The update operation may also output a message to a global reduction.
3. **edge** – Each edge with an input message processes it, possibly updates its state, and outputs a message to its destination node to be processed in the next graph step.

At most one operation occurs on each edge and at most one update operation occurs on each node. When multiple messages are pending to a node, they are processed as a single reduction operation. This means no race conditions can occur, and the programmer need not reason about relative timing of operations.

A sequential controller broadcasts messages to nodes to initiate the iteration. The broadcast value is fed to a node update operator at each receiving node. Graph steps may continue until the computation has quiesced and no messages are generated. For example, a graph relaxation usually only generates messages upon changes, so upon convergence there are no more messages (e.g. Bellman-Ford [Cormen et al. 1990]). Alternatively, the sequential component of the algorithm may decide when to end the iteration. For example, Conjugate Gradient [Hestenes and Stiefel 1952] uses a global reduce to decide when the error is small enough to stop.

2.1 Enabled Optimizations

In order to take advantage of GraphStep on a spatial implementation (e.g. FPGA or multicore processors) we must minimize communication work and latency and load balance memory, computation and communication to fit into small, distributed processing elements. To do this we use the exposed graph communication structure and exploit the associativity and commutativity properties of reduce operations.

2.1.1 *Placement for Locality.* The static graph structure is used to maximize the locality of neighboring nodes. The number of neighboring nodes placed into the same PE is maximized, and the distance between neighboring nodes in different PEs is minimized. This minimizes the volume of message traffic between processing elements. Since local communication is fast compared to cross-chip communication, it also minimizes each graph step's critical path latency due to message passing.

2.1.2 *Node Decomposition.* Load-balancing nodes into PEs must be performed to minimize the memory area per PE and minimize the computation work per PE. Often nodes with large numbers of neighbors inhibit load-balancing when scaling to many processing elements. Associativity and commutativity of reduction operations allows us to decompose a large node and distribute it across multiple processing elements (Figure 1). Node decomposition transforms a node with a large input-arity to a fanin tree of reduce operators followed by the state-holding root node. A node with a large number of outputs is decomposed into a root with a fanout tree to fanout messages. Note that knowledge of the structure of the graph is required to make connections from fanout tree leafs to fanin tree leafs.

2.1.3 *Static Scheduling.* The static graph structure is used to pre-schedule operation firings and message routes. Upon loading the graph, a static schedule is computed for a single graph step where each node and edge is active. Each PE and network switch inputs the pre-determined firing and routing choices from a dedicated memory. These

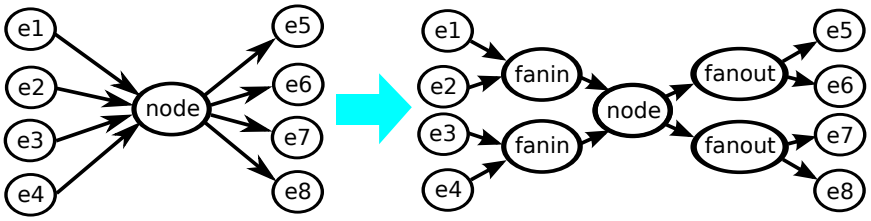

Fig. 1. Node decomposition

time-multiplexed memories store a data-independent, VLIW instruction sequence that is evaluated once per graph step.

Without knowledge of the static graph, a dynamic schedule must be computed online by (1) a packet-switched network to route messages and (2) extra PE control logic to fire operations. We find that static scheduling is typically more efficient than dynamic scheduling in terms of hardware area and time (Section 5). The dynamically scheduled case uses more hardware area than the statically scheduled case due to the high cost of packet-switched interconnect switches. Further, the static router performs offline, global routing to minimize network congestion. The static scheduler also combines the compute and communicate phases of each GraphStep.

2.1.4 *Hardware.* We can also specialize hardware to the GraphStep model. Node and edge operators and node and edge memories can be pipelined so each operator fires at the rate of one per cycle. To feed the operator pipeline messages must be input and output at the rate of one per cycle. Lightweight message handling is enabled by the GraphStep model since logic need not perform message ordering or buffer resizing.

Global broadcasts and reduces could be a significant source of latency since they must cross the entire machine. We map them to dedicated binary tree interconnect to eliminate potential congestion with other messages and eliminate latency due to interconnect switches.

2.2 Compute Model Comparison

This section explains how GraphStep differs from related concurrency models. GraphStep is high level in its domain, which reduces the detail the programmer must specify and manage. The compiler and runtime use the exposed communication structure to optimize for a spatial implementation.

Actors: In actors languages (e.g. Act1 [Lieberman 1987] and ACTORS [Agha 1998]), computation is performed with concurrently active objects that communicate via message passing. All computation is reduced to atomic operations that mutate local object state and are triggered by and produce messages. Similar to actors, GraphStep has the above restrictions. The primary difference is that actors programs are low-level descriptions of any concurrent computation pattern on objects, rather than a high-level description of a particular domain. The communication structure is, in general, dynamic and hence not visible to the compiler.

Streaming: Streaming persistent data-flow languages have a static or mostly static graph of operators that are connected by streams (e.g. Kahn Networks [Kahn 1974], SCORE [Caspi et al. 2000], Ptolemy [Lee 2005], Brook [bro 2004], Click [Shah et al.

2004]). These are used for high-performance applications such as packet switching and filtering, signal processing and real-time control. Like GraphStep, streaming data-flow languages are often high-level, domain-specific, and the spatial structure of a program can be used by a compiler. The primary difference is that in streaming computations the persistent nodes are operators given by the program, whereas in GraphStep the persistent nodes are data objects given by input to the program. For GraphStep the global graph steps free the implementation from the need to track an unbounded length sequence of tokens on each channel.

Data Parallel: Data parallelism [Blelloch et al. 1993; Koelbel et al. 1994; Hillis 1985; Dean and Ghemawat 2004] is a simple way to orchestrate parallel activity. A thread applies an operation in parallel to the elements of a collection. The operation may be applied to each element independently (map). It may also be a reduction or parallel-prefix operation (reduce) [Hillis and Steele 1986; Dean and Ghemawat 2004].

Machines that are entirely SIMD or have SIMD leaves [Hillis 1985; Habata et al. 2003; Lindholm et al. 2008]) are and important target for data-parallel languages. Like GraphStep, data-parallel programs can be very efficient since they map well to SIMD hardware. However, they do not typically describe operations on irregular data-structures efficiently and do not expose the communication structure of the application to the compiler.

Bulk Synchronous Parallel: BSP is an abstract model of parallel computers [Valiant 1990]. Programs written with a BSP library or language use barriers to synchronize between processors. Processors input messages from the last barrier-synchronized step and output messages to the next barrier-synchronized step. Unlike GraphStep, BSP programs do not expose the communication structure to the compiler.

3. APPLICATIONS

A variety of graph algorithms fit the GraphStep domain. We test the FPGA implementation performance of GraphStep on Bellman-Ford, Preflow-Push, and ConceptNet.

3.1 Iterative Numerical Methods

These are used for solving linear equations, finding eigenvalues and numerical optimization. In the examples Conjugate Gradient, Lanczos, and Gauss-Jacobi a commonly used Compressed Sparse Row representation for the matrix uses one node to represent a row of the matrix and one edge to represent a non-zero of the matrix. Each matrix-vector multiply is performed by one graph step, and each global dot-product is performed by a global reduce. The spatial implementation of sparse matrix-vector multiply from [deLorimier and DeHon 2005] achieved a speedup of an order of magnitude over highly tuned sequential implementations.

3.2 Graph Relaxation Algorithms

Algorithms in this sub-domain are composed of relaxation operations on directed edges. A relaxation operation on an edge updates its destination node's state based on its source node's state. If the destination node's state computed by the relaxation is less than its current state then its state changes. Every time the state of a node changes its out edges must relax. When there are no remaining relaxations, node states have reached a fixed point and the algorithm is finished.

If relaxations are timed improperly then there could be an exponential number of relaxation operations compared to the optimal timing. Synchronizing relaxations into graph

```
seq Graph {
  Node nodes; // nodes is a set of objects of type Node
  Node source; // source is a pointer to one object of type Node

  boolean bellman_ford() {
    unsigned nnodes = nodes.size();
    source.min_edge(0);
    // iterate until convergence (no nodes updated distance in the last iteration)
    //            or iter==nodes.size() in which case there is a negative cycle
    unsigned iter;
    unsigned active = true;
    for (iter = 0 ; active && iter < nnodes ; iter++) {
      // step is a primitive function which initiates one graph step and
      // returns true iff there are pending operation fires
      active = step();
    }
    // return true iff there is no negative cycle
    return iter < nnodes;
  }
}

node Node {
  Edge edges;
  float distance;

  // operate on two messages, bound to distance1 and distance2 respectively
  reduce tree min_edge (float distance1) (float distance2) {
    if (distance1 < distance2) return distance1;
    else return distance2;
  }
  update min_edge (float newdist) {
    if (newdist < distance) {
      distance = newdist;
      edges.propagate(distance);
    }
  }
}

edge Edge {
  Node to;
  float length;
  fwd propagate (float distance) {
    to.min_edge(distance + length);
  }
}
```

Fig. 2. Bellman-Ford code

App	Input	Nodes	Edges
Bellman-Ford	s38584.1	6448	20840
	s38417	6407	21344
	clma	8384	30462
	ex5p	1065	4002
	pdc	4576	17193
Preflow-Push	BVZ-tsukuba10_8	45273	143592
	BVZ-tsukuba10_4	90055	285220
	BVZ-tsukuba10_2	185388	591552
ConceptNet	small	14556	27275
	default	224876	553836

Fig. 3. Characteristics for Benchmark Graphs used with Sample Applications

steps bounds the number of graph steps by the number of nodes. Problems which may use relaxation algorithms include shortest-paths, finding a depth first search tree, identifying strongly connected components [Chandy and Misra 1982], global optimizations on program graphs [Kildall 1973], max-flow/min-cut, and constraint propagation in combinatorial problems like CNF SAT.

3.2.1 *Bellman-Ford.* Bellman-Ford is a shortest-paths algorithm [Cormen et al. 1990]. Each edge in the graph has a length, possibly negative. It finds the shortest path from a designated source node to all nodes, or detects the presence of a negative cycle. Each iteration takes a set of nodes whose distance from the source was updated on the previous iteration. Each out edge from the updated nodes is relaxed, which means that the shortest path so-far through it is checked against the shortest previously computed path to its destination node. If the new path is shorter, then the node updates its distance. The iteration continues until it quiesces, or until it detects a negative cycle.

Figure 2 shows Bellman-Ford in a high-level language for GraphStep. This high-level language expression does not automatically compile to hardware yet. Our algorithms are currently expressed in a slightly lower-level language that we expect will be an easy mapping target from this high-level language. For simplicity, the example omits code required to setup the graph.

We test Bellman-Ford as the kernel of register retiming of a circuit to find the minimum cycle time [Leiserson et al. 1983].

3.2.2 *Preflow-Push.* Preflow-push uses interactions between neighboring nodes to find the max-flow on a graph from a single source to a single sink [Cormen et al. 1990]. Preflow-push is a relatively more complex relaxation algorithm, that propagates updates through a graph. Unlike Bellman-Ford it always converges to a solution. It uses two basic operation types on nodes, and whether an operation can be applied to a node depends on its neighbors' states. The GraphStep algorithm cycles through four types of graph steps with different operations on nodes and edges in each.

We test Preflow-Push as the kernel of stereo vision problems [Boykov et al. 1998; Kolmogorov and Zabih 2001].

3.3 CAD Algorithms

CAD algorithms typically perform NP-hard optimizations on a circuit graph. Multilevel partitioning algorithms cluster nodes and iteratively reassociate them with partitions [Karypis

Component	Slices Each	Number	Slices
Total			1250K
Network total			411K
switch logic	28	992	28K
switch context memory	342	992	339K
channels			44K
PE total	410	2048	840K
logic	183	2048	375K
application memory	410	2048	840K
context memory	208	2048	426K

Table I. Area model for ConceptNet with the default graph using static hardware with 2048 PEs. Area is measured in terms of Virtex6 slices (see text).

and Kumar 1999]. Iterative placement algorithms move nodes to reduce cost functions with a random element to avoid local minima [Wrighton and DeHon 2003]. Parallel routing may perform shortest-path reachability searches on the circuit graph [DeHon et al. 2006]. The above placer and router are spatial implementations which act directly on the circuit graph and show orders of magnitude speedup over state-of-the-art sequential processor implementations. These were designed and implemented by hand, whereas GraphStep versions would automate much of the implementation work. Further, the router can use Bellman-Ford as its shortest-path kernel.

3.4 Semantic Networks, Knowledge-Bases and Databases

When these are represented as graphs, queries and inferences take the form of parallel graph algorithms, including marker passing [Fahlman 1979; Kim and Moldovan 1993], subgraph isomorphism, subgraph replacement, and spreading activation (e.g. ConceptNet [Liu and Singh 2004]).

3.4.1 *ConceptNet.* ConceptNet is a knowledge base for common sense reasoning compiled from a Web-based, collaborative effort to collect common sense knowledge [Liu and Singh 2004]. Nodes are concepts and edges are relations between concepts, each labeled with a relation-type. Spreading activation is a key operation for ConceptNet. Edges are given weights depending on their relation type. An initial set of nodes is chosen and each is given an activity of 1.0. Activities are propagated through the network, stimulating related concepts. After a fixed number of iterations, nodes with high activities are identified as the most relevant to the query.

4. IMPLEMENTATION

On a modern FPGA, the Virtex6, we can perform a memory operation in less than 3ns, a 16-bit add in less than 3ns, and send a signal across the distance of 2 PEs in the same 3ns cycle. However, we can place 512 PEs on today's largest Virtex6, meaning it takes over an order of magnitude longer (24 times) to communicate across the chip than to perform a local operation. Further Moore's Law scaling will allow us to place more PEs on a chip while maintaining fairly comparable relative delays such that cross chip communication will easily be two or more orders of magnitude greater delay than a local operation. These ratios of compute and communicate latency mean the location of computations matter.

We use specific area and timing costs from the Virtex6, but this general trend where cross-chip communication latencies exceed local computation costs by orders of magnitude

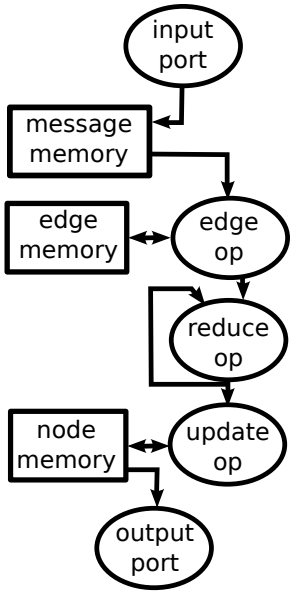

Fig. 4. Processing Element Dxatapath

```
// body executed per node
for i = 0 to nnodes
  emin = edgeoff[i]
  emax = edgeoff[i+1]
  r = reduce_id[i]
  j = emin
  // loop performs edge op, reduce op
1 while j < emax
2   m = input_message[j]
3   e = edge[j]
4   (f, n) = edge_op(e, m)
5   edge[j] = f
6   r = reduce_op(r, n)
7   j++
  // node update op
  a = node[i]
  (b, z) = update_op(a, r)
  node[i] = b
  j = emin
  // fanout output to out edges
8 while j < emax
9   sa = send_addr[j]
10  output_message[sa] = z
11  j++
```

Fig. 5. A sequential PE program that processes all input messages on one graph step and produces all output messages for the next graph step. The 11 instructions required per edge are numbered.

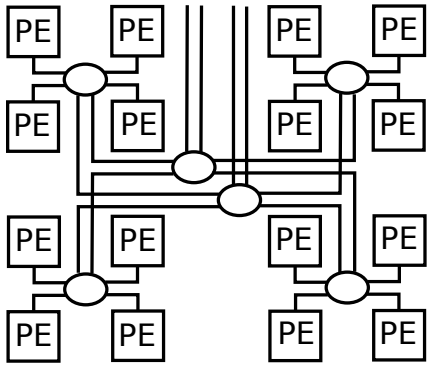

Fig. 6. Two stages of the BFT with 16 PEs and 4 channels up to the next stage

will be true of all high-performance silicon computations. To generate logic for the Virtex6 we use Synplify Pro 9.6.1 for synthesis and Xilinx ISE 10.1 for place and route.

4.1 Processing Element

PEs are designed for high throughput between operator logic and application memory and between the PE and the interconnect. In a spatial implementation logic is adjacent to

memory to enable high memory bandwidth. The communication centric approach requires a high message input and output rate. Typically a graph has many more edges than nodes so we provide dedicated node, edge and message memories, which allows the pipeline to handle one edge per cycle (Figure 4). The operations required by one edge are: at the source PE output a message, and at the destination PE input the message, perform the edge operation, and perform one iteration of the reduce operation. Operators are pipelined to perform one operation per cycle. Memories are dual ported to remove structural hazards between operations. On the output side, the node memory has one read per edge sent to. On the input side, the message memory has one read and one write per edge and the edge memory has one read and one write per edge. The node update operator fires only once per node so we allow it to conflict with edge sends.

The specialized datapath can be contrasted to a PE using an instruction set processor and a single local memory. Figure 5 shows the program for such a PE which, like the specialized hardware, requires one cycle edge operator, reduce operator or node operator and one memory word per node or edge. The sequential program has a throughput of one edge per 11 cycles.

For an efficient implementation the number of PEs must be large enough so memory area is comparable to logic area. Areas are measured in terms of Virtex6 slices, which each have 4 6-LUTs. Application memories are implemented with BlockRAMs, with one BlockRAM for every 83 slices. Table I accounts area used due to network components and PE logic and memory for the ConceptNet-default application (Table 3). PE area includes operators and controller logic, memory for the application, and memory for the static schedule. Total PE area is less than the sum of its components' areas since BlockRAMs and slices are separate hardware. Including interconnect, the area is 611 slices per PE, which allows a signal to cross 2 PEs per 3ns cycle.

4.2 Interconnect

The interconnect topology is designed to fit a two-dimensional spatial layout. Processing elements (PEs) are laid out in a grid, and connected with a Butterfly Fat-Tree (BFT) interconnect topology [Leiserson 1985]. Although a mesh topology would also correspond to two dimensions, the BFT is simpler to route. The BFT is constructed recursively, where the top level switches of a tree with n PEs connect the top level switches of four $n/4$ PE subtrees (Figure 6). The number of switches connecting two subtrees increases with the level of the tree in order to accommodate a larger number of cut graph edges between the two subtrees. The Rent parameter of the BFT, p, relates the number of PEs, n, in a subtree to the number of channels out of the subtree: $io\_channels = n^p$ [Landman and Russo 1971]. To fit the two dimensional hardware we set $p = 0.5$, so the number of channels out of an area scales with its perimeter. The fit to two dimensions allows us to layout a constant of one switch for every two PEs, regardless of tree size [DeHon 2000]. Further, the maximum PEs a signal crosses in a subtree with n PEs is $8\sqrt{n}$ PEs, which is proportional to the diameter of the subtree. The number of switches in the path is $\lg n$. Table I shows the area due to interconnect along with the area due to switch logic, memory for the static schedule, and the channels connecting switches.

5. IMPACT OF SPATIALLY AWARE OPTIMIZATIONS

This section uses our spatial hardware model to evaluate the benefit of the optimizations highlighted in Section 2.1. We evaluate the benefit of placement for locality, node decom-

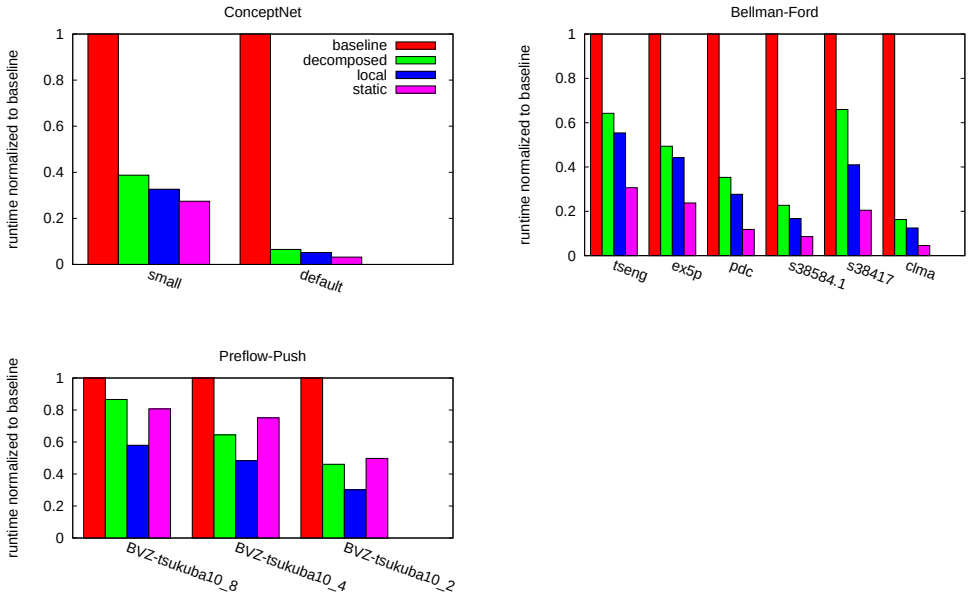

Fig. 7. Time of optimized implementations relative to the baseline. The first optimization decomposes nodes (decomp), the second also performs local placement (local), the third also schedules statically (static).

App	Graph	Baseline	Decomposed	Local	Static
ConceptNet	small	128	256	256	512
	default	256	2048	2048	2048
Bellman-Ford	tseng	256	128	128	128
	ex5p	256	256	128	256
	pdc	1024	1024	1024	512
	s38584.1	1024	1024	512	256
	s38417	1024	1024	512	2048
	clma	512	1024	1024	1024
Preflow-Push	BVZ-tsukuba10_8	2048	1024	2048	2048
	BVZ-tsukuba10_4	2048	1024	2048	2048
	BVZ-tsukuba10_2	1024	2048	2048	2048

Fig. 8. Number of PEs used by each application, graph and optimization

position, and compare the static scheduling option to dynamic scheduling.

5.1 Optimization Types

The baseline implementation places nodes of the original graph into PEs with the objective of maximizing the load balance between PEs. The load balancer takes the weight of a node to be the maximum of its input-arity and output-arity; since the PE processes one edge per cycle (Section 4.1) this is the approximate number of cycles spent on the node in one graph step.

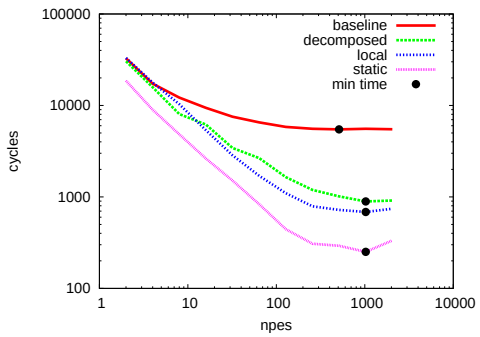

Fig. 9. Number of PEs and optimization to cycles, with the minimum cycle point marked

The first optimization uses the static graph structure to place neighboring nodes in the same PE or in nearby PEs in order to minimize total message volume and minimize the critical path of a graph step. It maximizes the locality of the placement while satisfying a load-balance constraint. At the top level of the BFT, graph nodes are partitioned into the two subtrees. The bipartition is chosen to minimize the number of cut edges between the two partitions while satisfying a load-balance constraint. Bipartitioning is applied recursively until nodes are placed in the leaf PEs. We use the UMpack's multi-level partitioner available from UCLA's MLPart5.2.14 [Caldwell et al. 2000].

The second optimization decomposes graph nodes with large input-arity or output-arity (Figure 1). Decomposition reduces the size of the largest nodes to allow scaling to a large number of small PEs. Each large node is transformed into a state-holding root with a fanin tree to perform reduce operations and a fanout tree to send messages. The input- and output-arity of all nodes is bounded to 64.

The third optimization performs static scheduling at load time. It then loads the schedule into switch and PE context memories. The unoptimized, dynamically scheduled implementation uses a packet-switched network to route messages as they are generated. Static scheduling removes the need for complex scheduling hardware and improves the quality of the schedule.

5.2 Optimization Results

Figure 7 shows, for the example applications, the cycles used relative to the baseline implementation with just decomposition applied (decomp), decomposition and placement for locality turned on (place), and decomposition, placement for locality and static scheduling turned on (static). The number of PEs used for each application and each optimization is chosen to minimize the total number of cycles, with a maximum of 2048 PEs (Table 9). Table 8 gives the number of PEs chosen for each application.

Figure 7 shows that all three optimizations together give speedups between 1.6 times (BVZ-tsukuba10_8) and 20 times (ConceptNet-default). Just decomposition achieves a speedup of 15 times for ConceptNet-default. Adding placement for locality gets a speedup of 1.6 for Bellman-Ford-s38417. Adding static scheduling gets a speedup of 2.7 times for Bellman-Ford-clma.

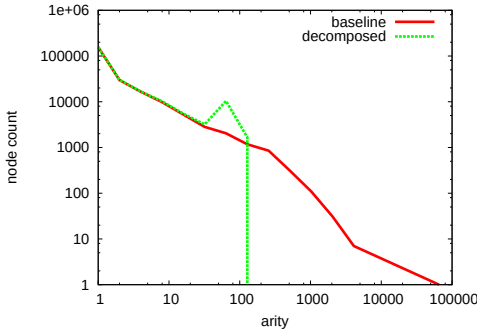

Fig. 10. Map from input- plus output-arity to node count for the baseline case and the decomposed case

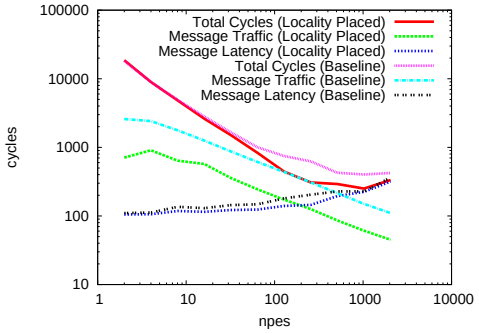

Fig. 11. Comparison between placement for locality and the baseline case showing the lower bounds on cycles per graph step due to chip crossing message latency and message traffic, along with the actual cycles

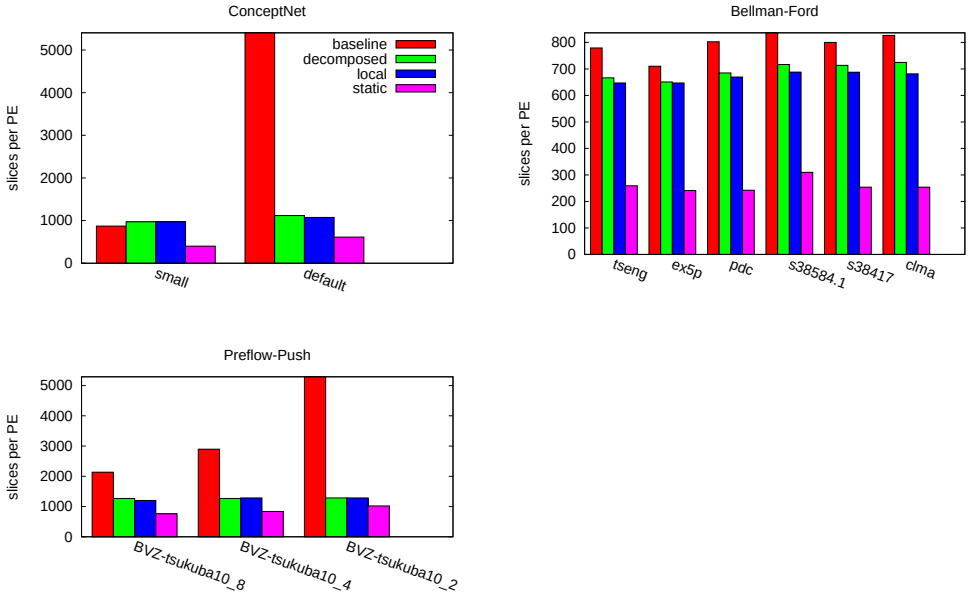

Fig. 12. Area in Virtex6 slices of a PE for each application and optimization.

Component	Slices Each	Number	Slices
Total			2199K
Network total			1109K
logic-switch	974	992	966K
channels			143K
PE total	532	2048	1090K
logic	183	2048	375K
app mem	532	2048	1090K

Table II. Area model for ConceptNet with the default graph with dynamic hardware with 2048 PEs. Area is measured in terms of Virtex6 slices (see Section 4).

5.2.1 *Node Decomposition.* Large nodes can prevent load balancing the computation due to fragmentation. The computation time for each node is proportional to its input-arity plus output-arity so the largest node imposes a lower bound on the cycles for each graph step. Figure 10 shows the result of decomposition on the distribution of node arities for the ConceptNet-default graph (Table 3). Before decomposition, the largest node has an arity of 52737, where the sum over all node arities is twice the number of edges: 1107672. Since the largest node in this case is about 1/20th the weight of all nodes, there can be little reduction in time by scaling above 20 PEs. After decomposition the largest node has an input- and output-arity of 64.

The benefit of decomposition increases as graph size increases because large, decomposed graphs can efficiently utilize more PEs than small, decomposed graphs. The benchmark graphs in Figure 7 are ordered by edge count from left to right for each application. For example, the speedup from decomposition for Bellman-Ford increases from 1.6 times for the tseng to 10 times for clma.

Decomposition is required often to fit all nodes into small PEs. Large PEs with fragmented memory use waste area and can increase computation time due to high chip-crossing latency. For ConceptNet-default decomposition reduces slices per PE by 5 times.

5.2.2 *Placement for Locality.* The primary effect of placement for locality is to decrease the message traffic. This can be seen in Figure 11 which compares the lower bound imposed by message traffic, the lower bound bound imposed by chip crossing latency, and the total time required for a graph step. At low PE counts, this shows the speedup available from increasing PE counts. Here we see that the locality-placed design significantly reduces the minimum cycles required for communications, avoiding this bottleneck and allowing greater performance as PE counts increase. Figure 11 further shows that performance at high PE counts is limited by communication latency, and this latency is only slightly impacted by placement for locality.

5.2.3 *Static Scheduling.* Figure 7 shows that static scheduling improves performance for ConceptNet and Bellman-Ford [Kapre et al. 2006]. This is because (1) the static scheduler can compute a higher quality route than the dynamic scheduler, given the same set of messages, (2) the static scheduler can combine the compute and communicate phases of each graph step, and (3) static hardware typically has lower area which decreases the chip crossing latency.

Table II gives areas for the dynamically scheduled hardware components for the ConceptNet-default application. It shows that the primary difference between dynamic and static hardware areas (Table I) is due to interconnect switch size. Figure 12 shows the Virtex6 slices

App	Graph	Activity
Preflow-Push	BVZ-tsukuba10_8	0.052395
	BVZ-tsukuba10_4	0.048740
Bellman-Ford	s38584.1	0.909322
	s38417	0.784112
	clma	0.861848
	ex5p	0.807392
	pdc	0.916647
ConceptNet	small	0.110513
	default	0.253764

Fig. 13. Activity factors for each application and graph

App	Graph	Sequential Time	GraphStep Time	Speedup
ConceptNet	small	10ms	7.2μs	1389
	default	490ms	31μs	15806
Bellman-Ford	tseng	72ms	3.8ms	19
	ex5p	68ms	3.3ms	21
	pdc	1.6s	13ms	123
	s38584.1	2.8s	29ms	97
	s38417	3.2s	21ms	152
	clma	6.9s	44ms	157
Preflow-Push	BVZ-tsukuba10_8	20s	23s	0.87
	BVZ-tsukuba10_4	100s	44s	2.3
	BVZ-tsukuba10_2	623s	106s	5.9

Fig. 14. Sequential and GraphStep runtimes for each application and graph

per PE for each application and optimization. The statically scheduled hardware area is about half the dynamic area. They plot the area per PE, where, as above, the number of PEs was chosen to minimize the total number of cycles.

However, static scheduling decreases performance for the Preflow-Push graphs tested. Table 13 show the average fraction of edges activated over graph steps. For Preflow-Push the low activation of approximately 1/20 causes a slow-down of less than 50

6. COMPARISON TO SEQUENTIAL PERFORMANCE

To evaluate the benefit of using GraphStep on spatial hardware we compare its runtime to sequential implementations of the GraphStep algorithms. Table 14 shows the results of the total runtime for the GraphStep implementation and a sequential implementation of each application studied. ConceptNet-default performs the most favorably with a speedup of 4 orders of magnitude. The larger Bellman-Ford graphs reach a 2 orders of magnitude speedup. The largest Preflow-Push graph reaches a 6 times speedup.

The sequential programs were run on a 3GHz Xeon. ConceptNet is implemented in C and compiled with gcc 4.3.2 using the -O3 option. It uses an active node queue to perform only the necessary updates on each iteration. Register retiming and its Bellman-Ford kernel are implemented in Ocaml and compiled with ocamlopt 3.10.2 using the -unsafe and -inline 2 options. Since activity for our Bellman-Ford graphs in close to 1 (Table 13), the implementation iterates over all nodes in the graph on each step. Preflow push is implemented in Ocaml and compiled with ocamlopt 3.10.2 using the -unsafe and

-inline 4 options. Each step of the outer iteration is analogous to a graph step and uses two queues to keep track of the active nodes. It uses efficient array based queues with O(1) time per push or pop operation.

7. CONCLUSION

To continue to turn the additional transistors provided by technology scaling into performance, we must exploit parallelism. Effective exploitation of this parallelism demands careful management of the location of computations so that fragmentation, communication latency and bandwidth requirements do not undermine the benefits of parallelism. Knowing the communication structure of a computation, we can automatically select the location of computations to minimize these costs and achieve efficient spatial implementations. Our GraphStep model captures this domain and exposes the communication structure to enable spatial optimizations without placing the burden of locality management on the programmer.

REFERENCES

2004. Brook project web page. Online: <http://brook.sourceforge.net>.

AGHA, G. 1998. *ACTORS: A model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge.

BLELLOCH, G. E., CHATTERJEE, S., HARDWICK, J. C., SIPELSTEIN, J., AND ZAGHA, M. 1993. Implementation of a portable nested data-parallel language. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. San Diego, 102–111.

BOYKOV, Y., VEKSLER, O., AND ZABIH, R. 1998. Markov random fields with efficient approximations. In *IEEE Conference on Computer Vision and Pattern Recognition*. 648–655.

CALDWELL, A., KAHNG, A., AND MARKOV, I. 2000. http://doi.acm.org/10.1145/368434.368864Improved Algorithms for Hypergraph Bipartitioning. In *Proceedings of the Asia and South Pacific Design Automation Conference*. 661–666.

CASPI, E., CHU, M., HUANG, R., WEAVER, N., YEH, J., WAWRZYNEK, J., AND DEHON, A. 2000. Stream computations organized for reconfigurable execution (SCORE): Extended abstract. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*. LNCS. Springer-Verlag, 605–614.

CHANDY, K. M. AND MISRA, J. 1982. Distributed computation on graphs: shortest path algorithms. *Commun. ACM 25,* 11, 833–837.

CORMEN, T., LEISERSON, C., AND RIVEST, R. 1990. *Introduction to Algorithms*. MIT Press.

DEAN, J. AND GHEMAWAT, S. 2004. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation*. USENIX.

DEHON, A. 2000. Compact, multilayer layout for butterfly fat-tree. In *Proceedings of the Twelfth ACM Symposium on Parallel Algorithms and Architectures (SPAA'2000)*. ACM, 206–215.

DEHON, A., HUANG, R., AND WAWRZYNEK, J. 2006. Stochastic spatial routing for reconfigurable networks. *Journal of Microprocessors and Microsystems 30,* 6 (September), 301–318.

DELORIMIER, M. AND DEHON, A. 2005. Floating-point sparse matrix-vector multiply for FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 75–85.

DELORIMIER, M., KAPRE, N., MEHTA, N., RIZZO, D., ESLICK, I., RUBIN, R., URIBE, T. E., KNIGHT, JR., T. F., AND DEHON, A. 2006. Graphstep: A system architecture for sparse-graph algorithms. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 243–151.

FAHLMAN, S. E. 1979. *NETL: A System for Representing and Using Real-World Knowledge*. MIT Press.

HABATA, S., YOKOKAWA, M., AND KITAWAKI, S. 2003. The earth simulator system. *NEC Res. & Develop. 44,* 1 (January), 21–26.

HESTENES, M. R. AND STIEFEL, E. 1952. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand. 49,* 6, 409–436.

HILLIS, W. D. 1985. *The Connection Machine*. MIT Press.

HILLIS, W. D. AND STEELE, G. L. 1986. Data parallel algorithms. *Communications of the ACM 29,* 12 (December), 1170–1183.

KAHN, G. 1974. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP CONGRESS 74*. North-Holland Publishing Company, 471–475.

KAPRE, N., MEHTA, N., DELORIMIER, M., RUBIN, R., BARNOR, H., WILSON, M. J., WRIGHTON, M., AND DEHON, A. 2006. Packet-switched vs. time-multiplexed FPGA overlay networks. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 205–213.

KARYPIS, G. AND KUMAR, V. 1999. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing 20,* 1.

KILDALL, G. A. 1973. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 194–206.

KIM, J.-T. AND MOLDOVAN, D. I. 1993. Classification and retrieval of knowledge on a parallel marker-passing architecture. *IEEE Transactions on Knowledge and Data Engineering 5,* 5 (October), 753–761.

KOELBEL, C. H., LOVEMAN, D. B., SCHREIBER, R. S., GUY L. STEELE, J., AND ZOSEL, M. E. 1994. *The high performance Fortran handbook*. MIT Press, Cambridge, MA, USA.

KOLMOGOROV, V. AND ZABIH, R. 2001. Computing visual correspondence with occlusions using graph cuts. In *IEEE International Conference on Computer Vision*. Vol. 2. 508–515.

LANDMAN, B. S. AND RUSSO, R. L. 1971. On pin versus block relationship for partitions of logic circuits. *IEEE Transactions on Computers 20*, 1469–1479.

LEE, E. 2005. Uc berkley ptolemy project. Online: <http://www.ptolemy.eecs.berkeley.edu/>.

LEISERSON, C., ROSE, F., AND SAXE, J. 1983. Optimizing synchronous circuitry by retiming. In *Third Caltech Conference On VLSI*.

LEISERSON, C. E. 1985. Fat-trees: Universal networks for hardware efficient supercomputing. *IEEE Transactions on Computers C-34,* 10 (Oct.), 892–901.

LIEBERMAN, H. 1987. *Concurrent object-oriented programming in Act 1*. MIT Press, Cambridge, MA, USA.

LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. 2008. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro 28,* 2, 39–55.

LIU, H. AND SINGH, P. 2004. Conceptnet – a practical commonsense reasoning tool-kit. *BT Technical Journal 22,* 4 (October), 211.

PIERCE, B. C. 2002. *Types and Programming Languages*. MIT Press.

SHAH, N., PLISHKER, W., RAVINDRAN, K., AND KEUTZER, K. 2004. NP-Click: A productive software development approach for network processors. *IEEE Micro 24,* 5 (September), 45–54.

VALIANT, L. G. 1990. A bridging model for parallel computation. *Communications of the ACM 33,* 8 (August), 103–111.

WRIGHTON, M. AND DEHON, A. 2003. Hardware-assisted simulated annealing with application for fast FPGA placement. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 33–42.

A. GRAPHSTEP SEMANTICS

To specify the formal semantics for GraphStep, we define a syntax, type-checking rules, static graph specification and rules, and state transition operational semantics. We use object-oriented terminology: classes specify object behavior, fields name object to object pointers and methods specify operations. The notation for program structure is in the style of [Pierce 2002].

Sequences are overlined, so $\overline{a} = a_1...a_i...a_n$. If two symbols are overlined together, (e.g. $\overline{c\ f}$) then they have the same index interval (e.g. $c_1...c_n$ and $f_1...f_n$). The bitvector data type is written as $[b_1...b_n]$. Nested bitvectors are considered flat, so $[[b_1, b_2], b_3] = [b_1, b_2, b_3]$.

A.1 Syntax

The syntax for a GraphStep program is:

P	::=	CLG; $\overline{\text{CLN}}$; $\overline{\text{CLE}}$	program
CLG	::=	`glob` c {$\overline{c\ f}$; $\overline{\text{MBCAST}}$; $\overline{\text{MGRED}}$}	global class
CLN	::=	`node` c {T; $\overline{c\ f}$; $\overline{\text{MNUP}}$; $\overline{\text{MNRED}}$}	node class
CLE	::=	`edge` c {T; $c\ f$; $\overline{\text{MEFWD}}$}	edge class
MBCAST	::=	`bcast` m (T) f m_{to}	global broadcast method
MGRED	::=	`gred` m (T) V OP	global reduce method
MNUP	::=	`nup` m (T, $\overline{\text{T}}$) $\overline{f}$ $\overline{m}$ OP	node update method
MNRED	::=	`nred` m (T) m_{to} OP	node reduce method
MEFWD	::=	`efwd` m (T, T) m_{to} OP	edge method
T	::=	B_x	data type is bitvector of length x
V	::=	$[\overline{0\|1}]$	value is bitvector
OP	::=	`op` $\{(T, T)\ g\}$	operator

Prefix keywords `glob`, `node`, and `edge` categorize classes into their *class kind*. Similarly, prefix keywords categorize methods into *method kinds*. Without loss of generality, the order of classes in a program and methods in a class are by kind. An operator specifies the function it performs as g, which is a function with fixed length bitvector input and output. The implementation of g is left unspecified.

A.2 Data Types

The typing rules for bitvectors include booleans, bitvector list, and append. The primitive operator is defined to have the correct type.

$$0 : \text{Bool} \qquad 1 : \text{Bool} \qquad \frac{b_1 : \text{Bool} \ldots b_n : \text{Bool}}{[\overline{b}] : B_n} \qquad B_i * B_j = B_{i+j} \qquad \frac{\texttt{op}\ \{(B_a, B_b)\ g\} \quad x : B_a}{g(x) : B_b}$$

A.3 Structure Access Functions

The following rules define functions to access class structure. *ctype*(c) is the state type of objects of class c, *classto*(f) is the class of the objects that field f can point to, and *class*(m) is the parent class of the method m. The sets C_{glob}, C_{node}, and C_{edge} categorize classes by kind and correspond to syntax tree nodes CLG, CLN, and CLE. Semicolons are used to denote multiple conclusions in an inference rule, or equivalently, multiple inference rules.

$$\frac{\texttt{glob}\ c\ \{\overline{c\ f}\,;\,\overline{m}\}}{c \in C_{glob}\ ;\ \textit{ctype}(c) = B_0\ ;\ \textit{classto}(f_i) = c_i\ ;\ \textit{class}(m_j) = c}$$

$$\frac{\texttt{node}\ c\ \{B_x\,;\,\overline{c\ f}\,;\,\overline{m}\}}{c \in C_{node}\ ;\ \textit{ctype}(c) = B_x\ ;\ \textit{classto}(f_i) = c_i\ ;\ \textit{class}(m_j) = c}$$

Additionally, *ffwd*(c) is an edge class's only field.

$$\frac{\texttt{edge}\ c\ \{B_x\,;\,c_{to}\ f_{to}\,;\,\overline{m}\}}{c \in C_{edge}\ ;\ \textit{ctype}(c) = B_x\ ;\ \textit{classto}(f_{to}) = c_{to}\ ;\ \textit{class}(m_j) = c\ ;\ \textit{ffwd}(c) = f_{to}}$$

The following rules define functions to access method structure. For a method m, *inmsg*(m) is its input message type, op_m is its operator, $fto(m)$ is the field to sends on, and $mto(m)$ is the method it sends to. The sets M_{bcast}, M_{gred}, M_{nup}, M_{nred}, and M_{efwd} categorize methods by kind and correspond to syntax tree nodes $MBCAST$, $MGRED$, $MNUP$, $MNRED$, and $MEFWD$. Each method is categorized by its kind into one of:

$$\frac{\texttt{bcast}\ m\ (B_x)\ f\ m_{to}}{m \in M_{bcast}\ ;\ \textit{inmsg}(m) = B_x\ ;\ \textit{mto}(m) = m_{to}\ ;\ \textit{fto}(m) = f}$$

$$\frac{\texttt{gred}\ m\ (B_x)\ \texttt{op}\ \{(B_a, B_b)\ g\}}{m \in M_{gred}\ ;\ \textit{inmsg}(m) = B_x\ ;\ \textit{op}_m = g}$$

A node update methods can send on multiple fields to multiple methods, so its *fto* and *mto* have subscripts to denote field selection.

$$\frac{\texttt{nup}\ m\ (B_x, \overline{B})\ \overline{f}\ \overline{m}\ \texttt{op}\ \{(B_a, B_b)\ g\}}{m \in M_{nup}\ ;\ \textit{inmsg}(m) = B_x\ ;\ \textit{op}_m = g\ ;\ \textit{fto}_i(m) = f_i\ ;\ \textit{mto}_i(m) = m_i}$$

$$\frac{\texttt{nred}\ m\ (B_x)\ m_{to}\ \texttt{op}\ \{(B_a, B_b)\ g\}}{m \in M_{nred}\ ;\ \textit{inmsg}(m) = B_x\ ;\ \textit{op}_m = g\ ;\ \textit{mto}(m) = m_{to}}$$

$$\frac{\texttt{efwd}\ m\ (B_x, B_y)\ m_{to}\ \texttt{op}\ \{(B_a, B_b)\ g\}}{m \in M_{efwd}\ ;\ \textit{inmsg}(m) = B_x\ ;\ \textit{op}_m = g\ ;\ \textit{mto}(m) = m_{to}\ ;\ \textit{fto}(m) = \textit{ffwd}(\textit{class}(m))}$$

A.4 Type Rules

The following rules infer *OK* iff a program is correctly typed. Programs and classes are *OK* if their methods are *OK*:

$$\frac{\text{CLG } \textit{OK}\ ;\ \overline{\text{CLN } \textit{OK}}\ ;\ \overline{\text{CLE } \textit{OK}}}{\text{P } \textit{OK}} \qquad \frac{\overline{\text{MBCAST } \textit{OK}}\ ;\ \overline{\text{MGRED } \textit{OK}}}{\texttt{glob}\ c\ \{\overline{c\ f}\texttt{;}\ \overline{\text{MBCAST}}\texttt{;}\ \overline{\text{MGRED}}\}\textit{OK}}$$

$$\frac{\overline{\text{MNUP } \textit{OK}}\ ;\ \overline{\text{MNRED } \textit{OK}}}{\texttt{node}\ c\ \{\text{T}\texttt{;}\ \overline{c\ f}\texttt{;}\ \overline{\text{MNUP}}\texttt{;}\ \overline{\text{MNRED}}\}\ \textit{OK}} \qquad \frac{\overline{\text{MEFWD } \textit{OK}}}{\texttt{edge}\ c\ \{\text{T}\texttt{;}\ c\ f\texttt{;}\ \overline{\text{MEFWD}}\}\ \textit{OK}}$$

Method rules restrict which kinds of methods can send to which others. They check that the type for a message is consistent between its sender and receiver, and that the class of the method sent to is the same as the class of the field sent on. They also check that the object state type and message types are consistent with the operator input and output types.

$$\frac{\textit{classto}(f) = \textit{class}(m_{to})\ ;\ m_{to} \in M_{nup}\ ;\ B_x = \textit{inmsg}(m_{to})}{\texttt{bcast}\ m\ (B_x)\ f\ m_{to}\ \textit{OK}}$$

$$\frac{B_a = B_x * B_x\ ;\ B_b = B_x}{\texttt{gred}\ m\ (B_x)\ \texttt{op}\ \{(B_a, B_b)\ g\}\ \textit{OK}}$$

$$\frac{\begin{array}{c} c = \textit{class}(m) \\ \textit{classto}(f_i) = \textit{class}(m_i)\ ;\ m_i \in M_{efwd} \cup M_{gred}\ ;\ B_{y_i} = \textit{inmsg}(m_i) \\ B_a = \textit{ctype}(c) * B_x\ ;\ B_b = \textit{ctype}(c) * B_n * B_{y_1} * ... * B_{y_n} \end{array}}{\texttt{nup}\ m\ (B_x, B_{y_1} ... B_{y_n})\ \overline{f}\ \overline{m}\ \texttt{op}\ \{(B_a, B_b)\ g\}\ \textit{OK}}$$

$$\frac{\begin{array}{c} \textit{class}(m) = \textit{class}(m_{to})\ ;\ m_{to} \in M_{nup}\ ;\ B_x = \textit{inmsg}(m_{to}) \\ B_a = B_x * B_x\ ;\ B_b = B_x \end{array}}{\texttt{nred}\ m\ (B_x)\ m_{to}\ \texttt{op}\ \{(B_a, B_b)\ g\}\ \textit{OK}}$$

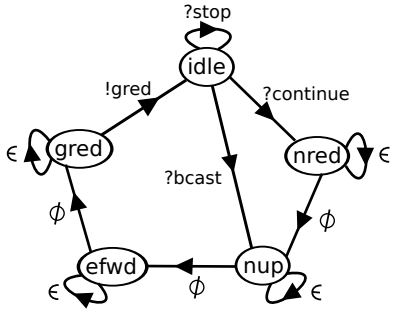

Fig. 15. State machine projected onto phase state component of entire state: $\{s_{idle}, s_{nup}, s_{efwd}, s_{gred}, s_{nred}\}$

$$\frac{\begin{array}{c} c = \mathit{class}(m) \\ \mathit{classto}(\mathit{ffwd}(c)) = \mathit{class}(m_{to}) \ ; \ m_{to} \in M_{nup} \cup M_{nred} \ ; \ B_y = \mathit{inmsg}(m_{to}) \\ B_a = \mathit{ctype}(c) * B_x \ ; \ B_b = \mathit{ctype}(c) * B_o * B_y \end{array}}{\texttt{efwd}\ m\ (B_x, B_y)\ m_{to}\ \texttt{op}\ \{(B_a, B_b)\ g\}\ \mathit{OK}}$$

Reduce operators should be associative and commutative. These rules are not checked by the type checker. Methods which are correct have the property *RUNOK*.

$$\frac{\begin{array}{c} \text{MK} \in \{\texttt{gred}, \texttt{nred}\} \\ g([x_1, x_2]) = g([x_2, x_1]) \ ; \ g([g([x_1, x_2]), x_3]) = g([x_1, g([x_2, x_3])]) \end{array}}{\text{MK}\ m\ (B_x)\ \texttt{op}\ \{(B_a, B_b)\ g\}\ \mathit{RUNOK}}$$

A.5 Graph Structure

The static graph is specified as:

O	:	the set of objects
$\mathit{class}(o)$	:	the class of object o
$o.f$	:	the set of objects pointed to by o on field f
o_g	:	the global object

Rules for the static graph are:

$\forall o, o', f : o' \in o.f \Rightarrow \mathit{class}(o') = \mathit{classto}(f)$	:	field pointer classes are consistent
$\forall o, f : class(o) \in C_{edge} \Rightarrow \lvert o.f \rvert = 1$	:	edges point to one object
$\lvert \{o \mid class(o) \in C_{glob}\} \rvert = 1$	:	there is one global object

A.6 State Transition Rules

The runtime state of a GraphStep process is defined by:

$R = (s, \mu, \gamma_{nup}, \gamma_{efwd}, \gamma_{gred}, \gamma_{nred}, \pi)$	:	entire state
$s \in \{s_{idle}, s_{nup}, s_{efwd}, s_{gred}, s_{nred}\}$	:	phase state
$\mu(o) : \mathit{ctype}(\mathit{class}(o))$	:	assignment of local state for objects
γ_{nup}	:	multiset of messages to a method in M_{nup}
γ_{efwd}	:	multiset of messages to a method in M_{efwd}
γ_{gred}	:	multiset of messages to a method in M_{gred}
γ_{nred}	:	multiset of messages to a method in M_{nred}
$\mathtt{msg}(o, m, v)$	:	a message for object o and method m with data v
$\pi(o)$	:	counts the number of reduce messages ready for object o

We make each γ_x a multiset so each send generates a unique message, even if there are others with the same (o, m, v) value. $\gamma_x \uplus \{\mathtt{msg}(o, m, v)\}$ denotes the multiset with one more instance of $\mathtt{msg}(o, m, v)$ than γ_x.

Computation is modeled as a sequence of state transitions. The GraphStep process communicates with a controller process which is not modeled here. Input and output values to the controller process are modeled as labels on state transitions. I/O state transition labels are:

?bcast(m, v)	:	Input global broadcast along method m with data v.
?continue	:	Input command to execute the next graph step.
?stop	:	Input command to stop graph steps.
!gred(v)	:	Output the result of the global reduce v.

Internally, the process executes a sequence of *phases*, and each phase contains a sequence of operation firings. Internal state transition labels are:

ϕ	:	Do an internal transition to the next phase.
ϵ	:	Do an internal transition in a phase.

Figure 15 shows the state machine projected onto the phase state component of the entire state.

Each state transition from R to R' with the label λ is written $R \xrightarrow{\lambda} R'$. Each of the following inference rules describes the transition that occurs when the premises of the inference are satisfied and the left hand state of the transistion matches the current state. The first graph step in an iteration is initiated with a transition from the idle state to the node update phase. $\mathbf{0}$ is the constant 0 function.

$$\frac{\gamma_{nup} = \{\mathtt{msg}(o, \mathit{mto}(m), v) \mid o \in o_g.\mathit{fto}(m)\}}{(s_{idle}, \mu, \emptyset, \emptyset, \emptyset, \emptyset, \mathbf{0}) \xrightarrow{\text{?bcast}(m,v)} (s_{nup}, \mu, \gamma_{nup}, \emptyset, \emptyset, \emptyset, \mathbf{0})}$$

Successive graph steps are initiated with a transition from the idle state to the node reduce phase:

$$(s_{idle}, \mu, \gamma_{nup}, \emptyset, \emptyset, \gamma_{nred}, \mathbf{0}) \xrightarrow{\text{?continue}} (s_{nred}, \mu, \gamma_{nup}, \emptyset, \emptyset, \gamma_{nred}, \mathbf{0})$$

Transition from the node reduce phase to the node update phase:

$$(s_{nred}, \mu, \gamma_{nup}, \emptyset, \emptyset, \emptyset, \pi) \xrightarrow{\phi} (s_{nup}, \mu, \gamma_{nup}, \emptyset, \emptyset, \emptyset, \pi)$$

Transition from the node update phase to the edge forward phase:

$$(s_{nup}, \mu, \emptyset, \gamma_{efwd}, \gamma_{gred}, \emptyset, \pi) \xrightarrow{\phi} (s_{efwd}, \mu, \emptyset, \gamma_{efwd}, \gamma_{gred}, \emptyset, \pi)$$

Transition from the edge forward phase to the global reduce phase:

$$(s_{efwd}, \mu, \gamma_{nup}, \emptyset, \gamma_{gred}, \gamma_{nred}, \pi) \xrightarrow{\phi} (s_{gred}, \mu, \gamma_{nup}, \emptyset, \gamma_{gred}, \gamma_{nred}, \pi)$$

Transition from the global reduce phase to the idle state: If there were no global reduce messages then output the nil bitvector. If there were global reduce messages then the transition occurs after all global reduce messages were reduce to a single message, and it outputs the result of the reduce.

$$(s_{gred}, \mu, \gamma_{nup}, \emptyset, \emptyset, \gamma_{nred}, \pi) \xrightarrow{!\text{gred}([\,])} (s_{idle}, \mu, \gamma_{nup}, \emptyset, \emptyset, \gamma_{nred}, \pi)$$
$$(s_{gred}, \mu, \gamma_{nup}, \emptyset, \{\texttt{msg}(o_g, m, v)\}, \gamma_{nred}, \pi) \xrightarrow{!\text{gred}(v)} (s_{idle}, \mu, \gamma_{nup}, \emptyset, \emptyset, \gamma_{nred}, \pi)$$

At the end of an iteration of graph steps clear the message state:

$$(s_{idle}, \mu, \gamma_{nup}, \emptyset, \emptyset, \gamma_{nred}, \pi) \xrightarrow{?\text{stop}} (s_{idle}, \mu, \emptyset, \emptyset, \emptyset, \emptyset, \mathbf{0})$$

The action of each method kind is specified by one of the following transition rules. Before we can define node update and edge forward rules we define the function *send*, which uses the message predicate $\overline{b}$ and message value $\overline{y}$ outputs produced by a method operator $op(m)$ to generate messages. Iff the bit b_i predicate is 1 then messages are sent on field f_i to method m_i with value y_i. The send on each field fans out to the set of objects, $o.f_i$.

$$\text{send}(o, \overline{f}, \overline{m}, \overline{b}, \overline{y}, M_{cat}) = \bigcup_i \begin{cases} \{\texttt{msg}(o', m_i, y_i) \mid o' \in o.f_i\} & \text{if } b_i = [1] \wedge m_i \in M_{cat} \\ \emptyset & \text{if } otherwise \end{cases}$$

A node update operation outputs messages on multiple fields to edge forward and global reduce methods. It modifies the local object state.

$$\frac{\begin{array}{c}\gamma'_{efwd} = \text{send}(o, \overline{\mathit{fto}(m)}, \overline{\mathit{mto}(m)}, \overline{b}, \overline{y}, M_{efwd}) \\ \gamma'_{gred} = \text{send}(o, \overline{\mathit{fto}(m)}, \overline{\mathit{mto}(m)}, \overline{b}, \overline{y}, M_{gred}) \\ \mathit{op}(m)([\mu(o), x]) = [y_s, \overline{b}, \overline{y}]\end{array}}{\begin{array}{c}(s_{nup}, \mu, \gamma_{nup} \uplus \{\texttt{msg}(o, m, x)\}, \gamma_{efwd}, \gamma_{gred}, \emptyset, \pi) \xrightarrow{\epsilon} \\ (s_{nup}, [o \mapsto y_s]\mu, \gamma_{nup}, \gamma_{efwd} \uplus \gamma'_{efwd}, \gamma_{gred} \uplus \gamma'_{gred}, \emptyset, \pi)\end{array}}$$

An edge forward operation outputs to a node reduce or node update method on a single field. It modifies local object state, and π for each node reduce destination is incremented.

$$\frac{\begin{array}{c}\gamma'_{nup} = \text{send}(o, \mathit{fto}(m), \mathit{mto}(m), b, y, M_{nup}) \\ \gamma'_{nred} = \text{send}(o, \mathit{fto}(m), \mathit{mto}(m), b, y, M_{nred}) \\ \pi'(o) = \pi(o) + |\{\texttt{msg}(o, m_c, v_c) \mid \texttt{msg}(o, m_c, v_c) \in \gamma'_{nred}\}| \\ \mathit{op}(m)([\mu(o), x]) = [y_s, b, y]\end{array}}{\begin{array}{c}(s_{efwd}, \mu, \gamma_{nup}, \gamma_{efwd} \uplus \{\texttt{msg}(o, m, x)\}, \gamma_{gred}, \gamma_{nred}, \pi) \xrightarrow{\epsilon} \\ (s_{efwd}, [o \mapsto y_s]\mu, \gamma_{nup} \uplus \gamma'_{nup}, \gamma_{efwd}, \gamma_{gred}, \gamma_{nred} \uplus \gamma'_{nred}, \pi)\end{array}}$$

A global reduce operation inputs two messages and outputs one:

$$\frac{op(m)([x_1, x_2]) = y}{\begin{array}{c}(s_{gred}, \mu, \gamma_{nup}, \emptyset, \gamma_{gred} \uplus \{\mathtt{msg}(o, m, x_1), \mathtt{msg}(o, m, x_2)\}, \gamma_{nred}, \pi) \xrightarrow{\epsilon} \\ (s_{gred}, \mu, \gamma_{nup}, \emptyset, \gamma_{gred} \uplus \{\mathtt{msg}(o, m, y)\}, \gamma_{nred}, \pi)\end{array}}$$

A node reduce operation inputs two messages and outputs one, and it decrements the π count state.

$$\frac{op(m)([x_1, x_2]) = y \ \mathbf{;}\ k = \pi(o) - 1}{\begin{array}{c}(s_{nred}, \mu, \gamma_{nup}, \emptyset, \emptyset, \gamma_{nred} \uplus \{\mathtt{msg}(o, m, x_1), \mathtt{msg}(o, m, x_2)\}, \emptyset, \pi) \xrightarrow{\epsilon} \\ (s_{nred}, \mu, \gamma_{nup}, \emptyset, \emptyset, \gamma_{nred} \uplus \{\mathtt{msg}(o, m, y)\}, \emptyset, [o \mapsto k]\pi)\end{array}}$$

When there is only one reduce message to an object, it is changed to an update message.

$$\frac{\pi(o) = 1}{\begin{array}{c}(s_{nred}, \mu, \gamma_{nup}, \emptyset, \emptyset, \gamma_{nred} \uplus \{\mathtt{msg}(o, m, x)\}, \pi) \xrightarrow{\epsilon} \\ (s_{nup}, \mu, \gamma_{nup} \uplus \{\mathtt{msg}(o, m, x)\}, \emptyset, \emptyset, \gamma_{nred}, \pi)\end{array}}$$

The following rules hide internal transitions by folding them into input and output transitions. First the closure of atomic operation (ϵ) transitions is included in phase (ϕ) transitions. Then the closure of phase transitions is included in i/o transitions. λ is a variable over input and output transition labels.

$$\frac{R \xrightarrow{\phi} (\xrightarrow{\epsilon})^* R'}{R \xrightarrow{\phi} R'} \qquad \frac{R \xrightarrow{\lambda} (\xrightarrow{\phi})^* R'}{R \xrightarrow{\lambda} R'} \qquad \lambda \neq \phi, \epsilon$$