Programming Model and Architectural Needs for Graph Applications on Continuum Computing Architecture

Bibrak Qamar Chandio

bchandio@iu.edu

School of Informatics, Computing and Engineering

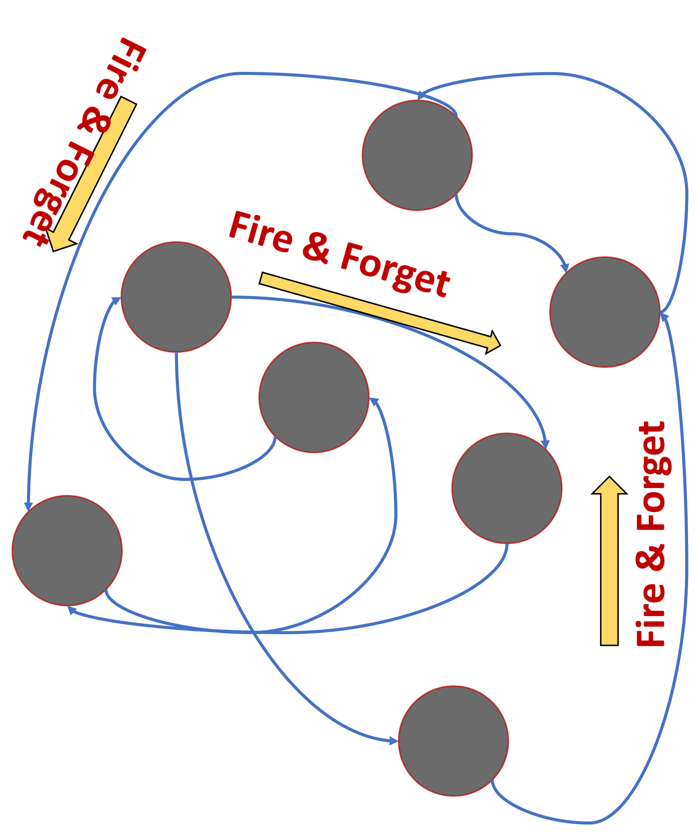
Indiana University Bloomington

Abstract

High Performance Computing hardware, which offers large amount of parallelism, is grounded in architectural and technology assumptions that practically impose limits on granularity of the parallelism. One such factor is Bulk Synchronous Parallel (BSP) model. To elevate this one promising exploration space is fine grain event driven execution models, such as ParalleX. This work explores Graphs---structures that inherently have large amount of parallelism. Asynchronously processing graphs using event driven execution models exposes this inherent fine grain parallelism. We implement asynchronous graph processing algorithms, especially the Single Source Shortest Path (SSSP) under HPX runtime system (which is an implementation of the ParalleX model) and concurrent priority queues, notably SprayList, Lotan and Shavit, and Linden and Jonsson Priority Queue. Performance results reveal insights into aspects of execution, specially the relationship between dynamic growth of work and scheduling policy and the overhead that is introduced by scheduling policies. From this experience we abstract out primitives that are need for asynchronous graph processing. These primitives will later help in defining an Instruction Set Architecture (ISA) for a graph based memory accelerator. The ISA will contribute towards Continuum Computer Architecture (CCA)—a family of non-von Neumann architectures that combine parallel control flow semantics of the ParalleX execution model with homogenous highly replicated lightweight compute cells.

Asynchronous Graph Processing

Graph processing generally involves low FLOP to Byte ratio and irregular data access pattern. This when mapped to Bulk Synchronous Model (BSP), which assumes some useful amount of granularity of computation before communication phase, leads to under exploitation of the large inherent parallelism that is naturally available in graph structures.



We employ asynchronous message driven computations to exploit this parallelism.

Thinking like a Vertex, asynchronously:

- Vertex becomes active by calling a vertex function
- When active a vertex can make neighboring vertices active by sending an active message—Diffusion.
- This message is asynchronous and *fire-and-forget.*
- Note there is no predetermined or runtime DAG because there could be cycles in the graph. Termination detection problem.
- We implement Dijkstra–Scholten algorithm for termination detection. It creates an implicit spanning tree of the graph and it naturally unfold when computation finishes.

Proposed HPX Diffuse

```
hpx_diffuse(vertex_id, vertex_func, terminator, predicate,
            arg_1, arg_2, …, arg_n);
```

1. `vertex_id` is the memory location at which the vertex exists: we send the parcel there
2. `vertex_func` is the active message that contains the vertex program (the diffusing computation)
3. `terminator` is an object that knows if this diffusion is finished or not
4. `predicate` is the invariant if false returns from the vertex_func without generating new work
5. `arg_1` to `arg_n` are any arguments to the vertex_func

Our HPX5 Implementation of SSSP	SSSP Implemented using Diffuse Primitive
(code below)	(code below)

Our HPX5 Implementation of SSSP

```
HPX_ACTION(sssp_vertex_func);
sssp_vertex_func_handler (hpx_addr_t vertex,
                          hpx_addr_t Graph,
                          hpx_addr termination_and_LCO,
                          int incoming_distance_from_root)
{
  hpx_lco_sema_p(v->vertex_local_mutex);
 // some part of termination detection logic might go here

  if(vertex.my_distance > incoming_distance_from_root){
    //update
    vertex.my_distance = incoming_distance_from_root;

    //diffuse
    for-all u in neighbors {
      int new_distance = u.weight+vertex.my_distance;
      hpx_call(u, sssp_vertex_func, Graph,
               termination_and_LCO, new_distance);
       // some part of termination detection logic might
       // go here
    }
  // some part of termination detection logic might
  // go here: i.e ack back to the sender
  hpx_lco_sema_v(vertex.vertex_local_mutex, HPX_NULL);
}
```

- Predicate logic hidden from RTS
- Termination Detection Logic embedded in the user code
- Explicitly insuring that only one action is running on a single vertex

SSSP Implemented using Diffuse Primitive

```
HPX+_ACTION(sssp_vertex_func);
sssp_vertex_func_handler (hpx_addr_t vertex,
                          hpx_term terminator,
                          hpx_addr_t Graph,
                          int incoming_distance_from_root)
{
  //update
  vertex.my_distance = incoming_distance_from_root;

  //diffuse
  for-all u in neighbors {
    int new_distance = u.weight+vertex.my_distance;
    hpx_predicate predicate = new Predicate(hpx_there >
                                            new_distance);
    hpx_diffuse(u, sssp_vertex_func, terminator, predicate,
                Graph, new_distance);
  }// for-all ends
} // vertex function ends
```

- Predicate logic orchestrated and RTS can see it → new opportunities
- Termination Detection Logic provided by RTS → new opportunities
- RTS insures only one action is running on a **read-write** (vertex) object
- Less lines of code → programmer productivity

HPX

High Performance ParalleX (HPX) is a distributed Asynchronous Many Task (AMT) runtime system.

- Scalable global name space
- Processes that span multiple nodes
- Preemptive threads as first class objects incorporating fine-grain data flow parallelism
- Powerful but lightweight synchronization primitives such as dataflow and futures constructs, called LCOs.
- Avoids the use of locks and/or barriers in parallel computations
- Reads and writes on LCOs are globally atomic and require no other synchronization mechanism.
- Move work to data (active message).

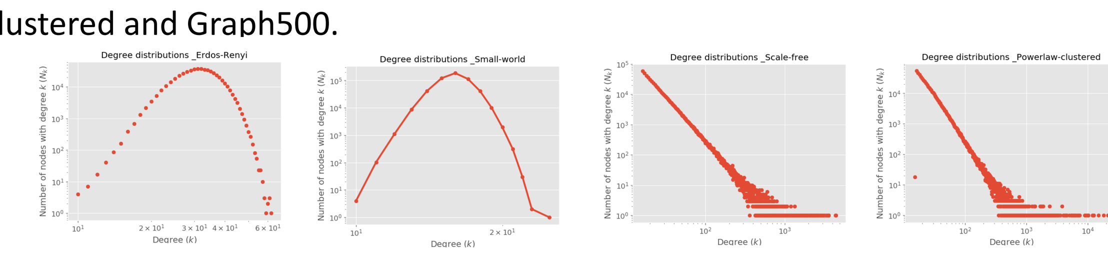
Experimental Method and Data-set Details

We report the time to solution and the number of actions (dynamic work) done. Here action is the active message generated at runtime. In an ideal run SSSP should traverse a single edge just once, therefore we divide it with the number of edges of the graph and report Actions Normalized.

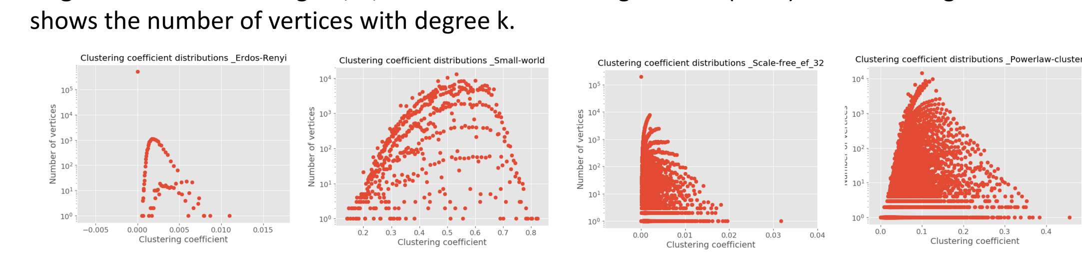
Number of Async SSSP implementations: 2 (HPX-SSSP and Conc-SSSP [1])

We ran our HPX-SSSP code and the Conc-SSSP (using concurrent priority queues) on the Indiana University's Bigred2 system, where each node is 32 cores.

There are 5 different Graphs: Erdos-Renyi, Small-World, Scale-Free, Powerlaw-Clustered and Graph500.

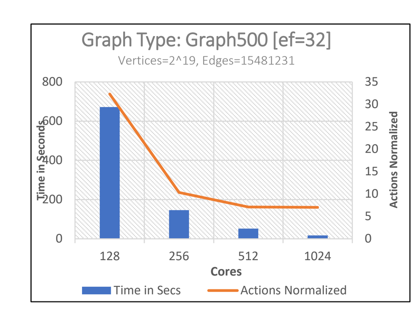


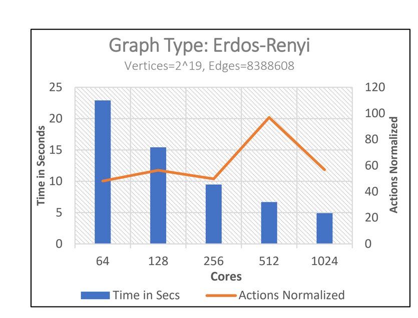
Degree Distribution: degree, k, is the number of edges from (or in) a vertex. Degree distribution shows the number of vertices with degree k.



Clustering Coefficient: degree to which vertices in a graph tend to cluster together

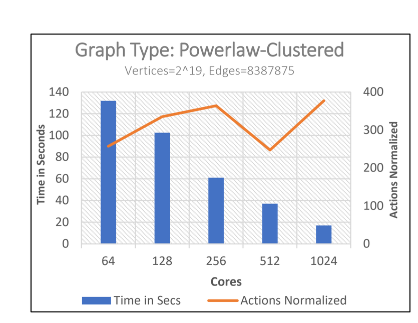
Performance Measurement Results

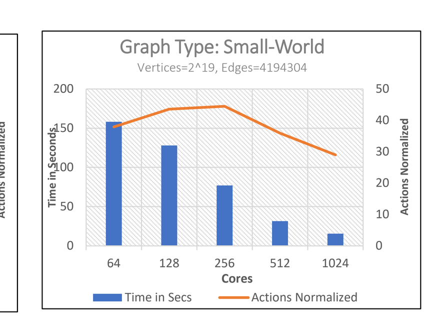


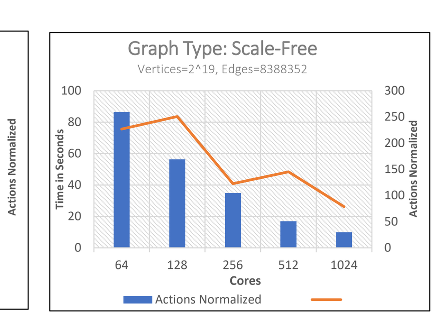


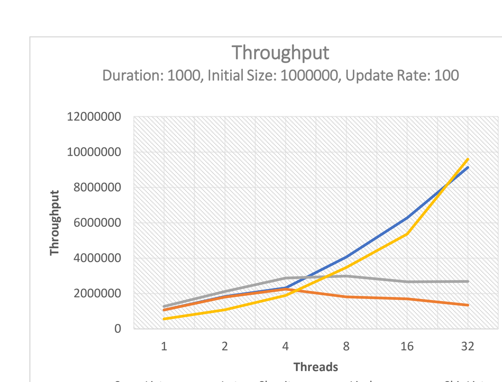
HPX

- In general as we increase HPX processes time to solution decreases.
- This comes not only from the added parallelism but also the dynamic nature of the computation.
- As we increase HPX processes we increase number of LIFO queues (each HPX process has its own LIFO queue) this in turn creates some randomness, which help do better scheduling.
- We also find that our computation is transcendental in the sense of scaling.
- As we increase number of processes (cores) the work per process, depending on runtime nature and graph structure, will grow or shrink, thereby it can either behave as strong scaling or weak scaling.



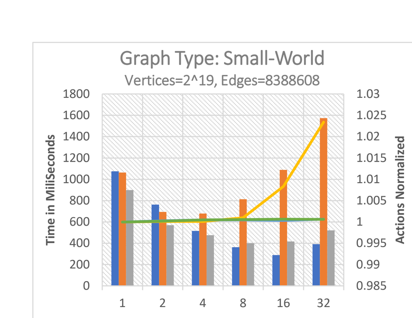


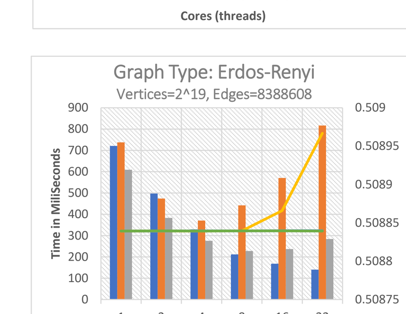


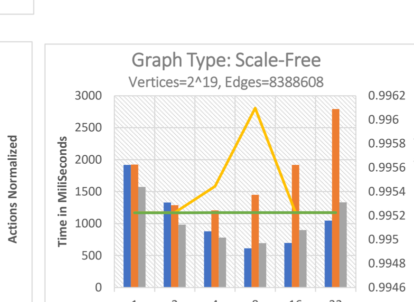


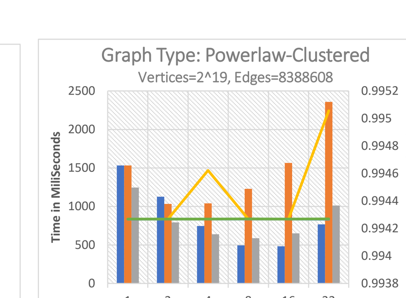
Concurrent Priority Queue SSSP

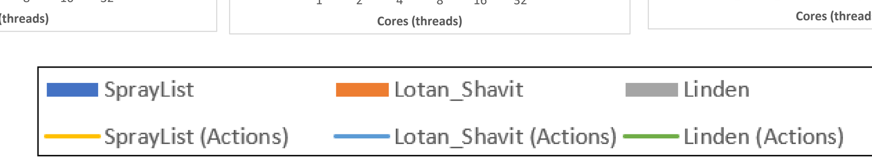
- SprayList performed the best overall with Linden in the second spot.
- Lotan_Shavit does not scale better.
- Throughput for each data-structure can be responsible for their respective scalability, or lack thereof.
- Since these are priority structures they do not create more work, almost all are under ideal work, i.e. they don't traverse more than number of edges. Since this is a shared memory implementation it gives the best case of a single queue for the entire graph operation, where threads can use push model in conjunction.
- Note: In a distributed setting this will not be the case since we will have multiple independent priority queues, therefore computation might not relax that easily. This shows best of the world and will help as a guide in making trade-offs.











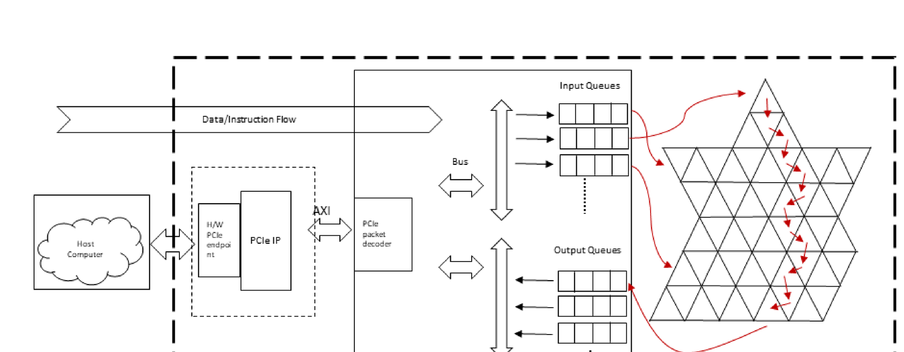
Graph Processing and the Continuum Compute Architecture: Future Work

CCA

- Continuum Compute Architecture is a new class of non von Neumann architectures.
- Offers fine grain parallelism
- Small compute cells organized such that it creates active memory.

Future Work

This work will help guide design of programming interface to CCA and through that it will help in extracting out basic primitives that need to be implemented in CCA for graph processing.



References:

[1] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The SprayList: a scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP 2015). ACM, New York, NY, USA, 11-20. DOI: https://doi.org/10.1145/2688500.2688523

[2] T. Sterling, D. Kogler, M. Anderson, and M. Brodowicz. SLOWER: A performance model for exascale computing. Supercomputing frontiers and innovations, 1(2), 2014.