Exploring Asynchronous Graph Processing

Spring 2018 Report

Bibrak Qamar Chandio [*]

May 03, 2018

1 Summary

Unlike a predetermined computational dependency graph [1] or a one that is determined at runtime[2], in asynchronous graph processing, there is no dependency graph. This is primarily because of arbitrary structure of the input graph with arbitrary weights, whereby the execution (program) relies on some arbitrary invariant to advance. Listing 1 shows such a program the Single Source Shortest Path (SSSP). This makes it difficult to know the amount of work that will be performed at runtime, since the execution can potentially take any path to the solution(s). There is another implicit problem with such computations called the Termination Detection Problem, which does not manifest in formulations in Bulk Synchronous Parallel (BSP) counterparts.

This report primarily talks about asynchronous graph processing along with its implementation on an asynchronous event driven runtime system called HPX[1]. It also sheds some light on some preliminary benchmark results. It also makes some suggestions for runtime system developers as to functionalities that will increase programmer productivity for asynchronous graph processing and potentially making the computation fast. Finally this report also talks about the same problem but implemented using concurrent priority data-structures (specially the SprayList [3]). Although not a purist approach[2] since the concurrent data-structures are devised to be run and scale on contemporary architectures but nevertheless it presents a good contrast as to what the state-of-the-art approaches on contemporary architectures reap for asynchronous graph processing.

[*]Intelligent Systems Engineering, Indiana University Bloomington

[1]It must be noted that HPX is built on top of LIFO queues therefore it doesn't do intelligent scheduling of parcels.

[2]A non Von Neuman approach that naturally fits graph processing like CCA[4]

```

diffuse(vertex v, int distance):
  if v.distance >= distance:
    v.distance = distance
    for u in v.neighbors:
      diffuse(u, v.distance + u.weight)

SSSP(vertex src):
  src.distance = 0
  for u in src.neighbors:
      diffuse(u, u.weight)
```

Listing 1: Asynchronous SSSP

2 Asynchronous Graph Processing

Graph structures contain massive amounts of inherent parallelism. Processing graphs in level Bulk Synchronous Parallel limits the parallelism to graph frontiers thereby leaving the natural parallelism to go waste. Asynchronously processing the graph mitigates this but comes at the cost of chaotic behavior, which leads to wasted work. It also comes at the cost of low network bandwidth utilization on current network fabrics. In this report we try to understand/quantify this chaos. We implement asynchronous Single Source Shortest Path (SSSP) algorithms, for two execution regimes, HPX and concurrent priority queues. HPX implementation scales across multiple NUMA domains whereas the concurrent priority queues, that we investigate, are limited to a single shared memory system. Both implementations are described in detail in Section 3 and Section 4, respectively.

2.1 Diffusing Computation

Asynchronous graph processing can be best described as a diffusing computation where:

I Each Vertex (in the graph) becomes active by calling a vertex function.

II When active, a vertex can make neighboring vertices active by sending a message, i.e. the diffusion.

III On receiving the message, a vertex can check some predicate (condition) on itself from the message data and decide whether to activate itself. This is where relaxation and scheduling comes.

IV This diffusion message is asynchronous with fire-and-forget semantics, which implies no DAG because there could be cycles in the graph.

V At some point when there is no work to do the vertex deactivates.

VI The whole diffusion computation finishes when there is no vertex active and there is no message in transit.

Figure 1 shows an example of diffusing computation that performs SSSP.

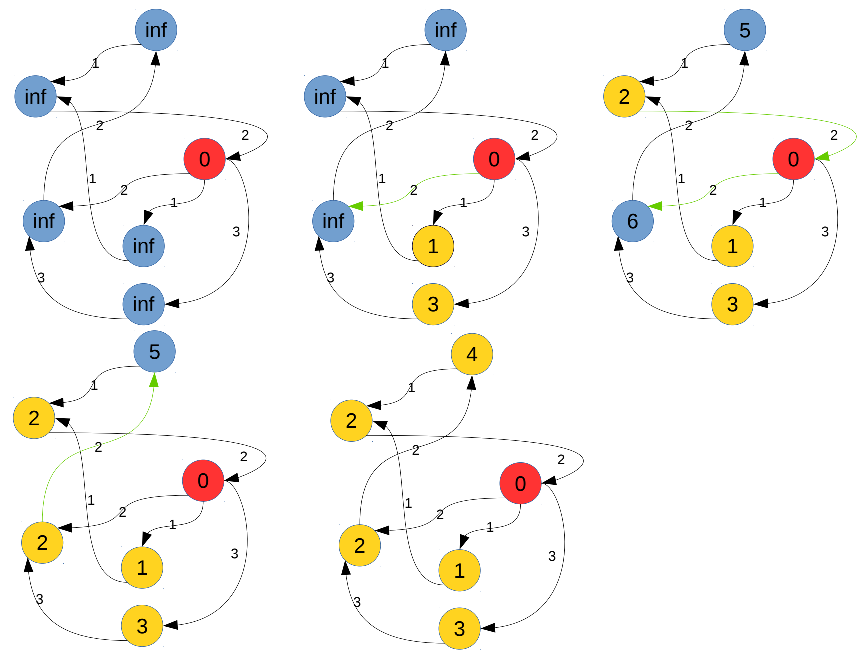

Figure 1: Asynchronous SSSP for a graph of size 6. Red indicates the source vertex: Yellow indicates relaxed vertex (monotonically decreasing with value indicated inside the vertex circle): Blue indicates never visited: Black arrow indicates edges: Green arrow indicates edge that is active, meaning a message is in transit. Note the edges are weighted representing distances.

3 HPX Implementation

We implemented asynchronous SSSP using HPX-5. Listing 2 shows pseudocode for SSSP HPX-5 implementation. We extensively used HPX's actions to perform vertex computations. This meant that very large number of small messages were generated on the network. For the purposes of termination detection we used Dijkstra–Scholten algorithm [5], which creates an implicit spanning tree. As computation comes to end the tree naturally unfolds and becomes empty, signifying termination. This comes at a cost of extra acknowledgment message for each diffusion massage.

```

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);
}
```

Listing 2: Asynchronous SSSP on HPX5

In principle this strategy hides latency by exposing and exploiting the inherent parallelism. In practice, for graph processing the nature of tasks (vertex actions) is too fine grain. Therefore on contemporary architectures like the Bigred II[6], the performance suffers since the network is not designed to scale well for large amounts of small messages.

Graph Scale	Vertices	Edges	Ideal actions
19	524286	15481231	30962462
Cores	Time in Secs	Actions Invoked	Actions Normalized
128	678.770931	1027155159	33.1742
256	155.189979	379368634	12.2525
512	60.4853586	272983141	8.81658
1024	28.969467	285339900	9.21567

Table 1: Performance results of SSSP on HPX-5

Table 1 shows the performance results on a scale 19 (2^{19} vertices) graph from the Graph500 generator. Here ideal actions represent the minimum number of HPX actions needed to find the SSSP. Theoretically it means to traverse a single edge only once. We add acknowledgment actions to this to get the total number of ideal actions. This helps in finding actions normalized, which gives an idea of how much extra work is performed. How much extra work gets to be performs mostly depends on two important factors, the scheduling policy and the chaotic runtime behavior.

In our case we had no scheduling policy since HPX-5 does not support priority queues for its scheduling of parcels (actions). Instead it naively uses a LIFO queue. The reason there is a increase in performance from small number of cores to large is due to the fact that as we increase HPX process we also increase the queues. Each HPX process has its own queue. This increase leads to more randomness as compared to LIFO-ness and this randomness in turn reaps some performance.

3.1 Some Recommendations

From this experience we found three main areas of improvement for HPX-5 that could lead to better performance and programmer productivity. These include:

I Orchestrating predicate logic so that HPX scheduler gets access to it and schedule actions intelligently.

II Providing termination detection logic with in HPX so that it makes code expression easy thereby not only increasing programmer productivity but also potentially increasing performance.

III Providing read-write access labels for memory regions will eliminate programmer's needs for explicitly locking the HPX process/thread on vertex object. This will increase programmer productivity and also expose new scheduling opportunities for HPX.

To this end we propose hpx_diffuse call. This captures the basics of diffusing computation that include the vertex function, the predicate that helps in scheduling and the termination detector.

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

- **vertex_id** the memory location at which the vertex exists: we send the parcel there
- **vertex_func** the active message that contains the vertex program (the diffusing computation)
- **arg_1 to arg_n** any arguments to the vertex-func
- **terminator** is the object that knows whether this diffusion is finished
- **predicate** the invariant, if false returns from the vertex_func without generating new work

Listing 3 shows hpx_diffuse in action and how our previous HPX-5 SSSP implementation, Listing 2, could look like.

```

HPX+_ACTION(sssp_vertex_func);

sssp_vertex_func_handler (hpx_addr_t vertex, hpx_addr_t Graph,
                              int incoming_distance_from_root,
                               hpx_term terminator) {
```

```
  //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, Graph, new_distance,
                terminator, predicate);
  }// for-all ends
} // vertex function ends
```

Listing 3: Asynchronous SSSP on HPX5 using hpx_diffuse

4 Priority Queue Implementation

In this section we show results for SSSP implemented using concurrent relaxed priority queues. Although current implementation is only for shared memory systems but nevertheless it gives a good glimpse into the nature of asynchronous fine grain task scheduling and execution. It basically stripes out all the HPX related overhead and rigidity of the LIFO queue.

Figure 2 shows results for a scale 21 (2^{21} vertices) graph from the Graph500 generator. We use three priority queues namely SprayList Priority Queue[3], Lotan and Shavit Priority Queue[7] and Linden and Jonsson Priority Queue[8]. Graph500 MPI reference implementation results are also presented for comparison.

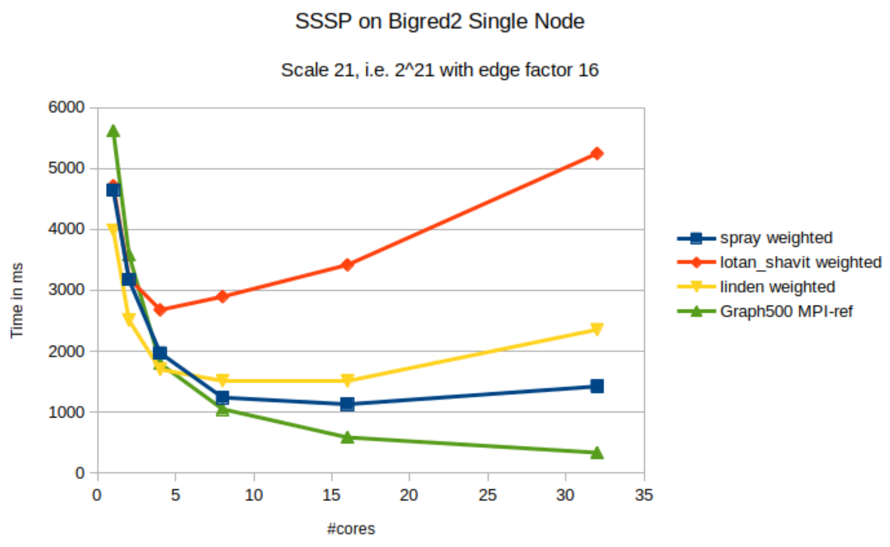

Figure 2: Asynchronous SSSP for scale 21 graph using concurrent priority queues. Each p-thread was pined to a single unique core. Note time is in milliseconds.

The reason lotan_shavit and linden don't scale well is due to their poor throughput as depicted by the throughput experiment in Figure 3. In this experiment there were 50% read and 50% update (i.e. deletemin) operations.

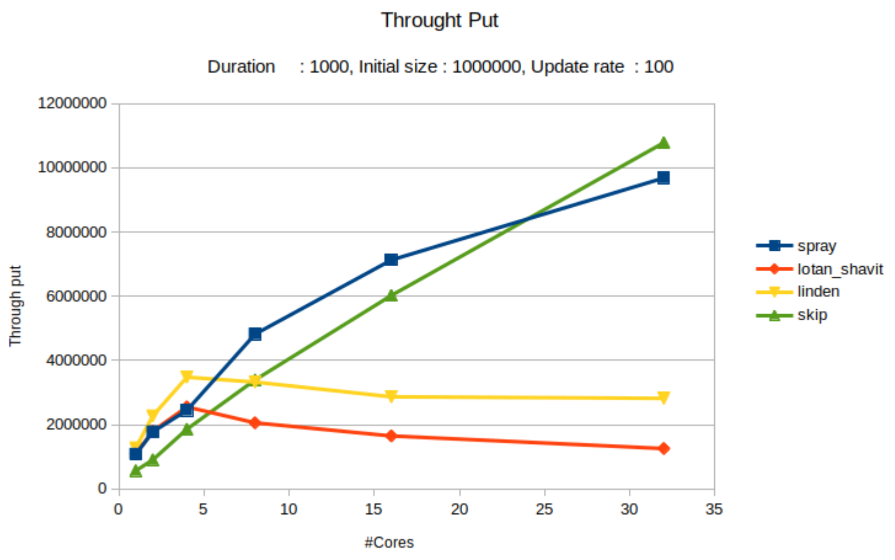

Figure 3: Throughput of various concurrent datastructures for varying thread count on Bigred II. Note skiplist is not a priority queue since it just removes a random value. It is here for reference purposes.

5 Future Work

From this experience, in the future I intend to implement distributed memory version of SSSP using concurrent priority queues. This will give a bare infrastructure for asynchronous graph processing since it will stripe of overheads associated with big runtime systems like HPX-5. It will help understand the limits of asynchronous graph processing on current system architectures. It will also help in the co-design of a memory accelerating hardware that is conceived from ground up to be for asynchronous graph processing such as CCA[4].

References

[1] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.

[2] Qingyu Meng, Alan Humphrey, John Schmidt, and Martin Berzins. Investigating applications portability with the uintah dag-based runtime system on petascale supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 96:1–96:12, New York, NY, USA, 2013. ACM.

[3] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: A scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 11–20, New York, NY, USA, 2015. ACM.

[4] Maciej Brodowicz and Thomas Sterling. A non von neumann continuum computer architecture for scalability beyond moore's law. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, pages 335–338, New York, NY, USA, 2016. ACM.

[5] Sukumar Ghosh. *9.3.1 The Dijkstra–Scholten Algorithm ,Distributed Systems: An Algorithmic Approach.* CRC Press, 2010.

[6] Bigred II. `https://kb.iu.edu/d/bcqt/`. Accessed on May-03-2018.

[7] N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 263–268, 2000.

[8] Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In Roberto Baldoni, Nicolas Nisse, and Maarten van Steen, editors, *Principles of Distributed Systems*, pages 206–220, Cham, 2013. Springer International Publishing.