Naiad: A Timely Dataflow System

FAN Junbo
1155076720

0. Introduction

1. Timely Dataflow

2. Implementation

3. Writing Programs on Naiad

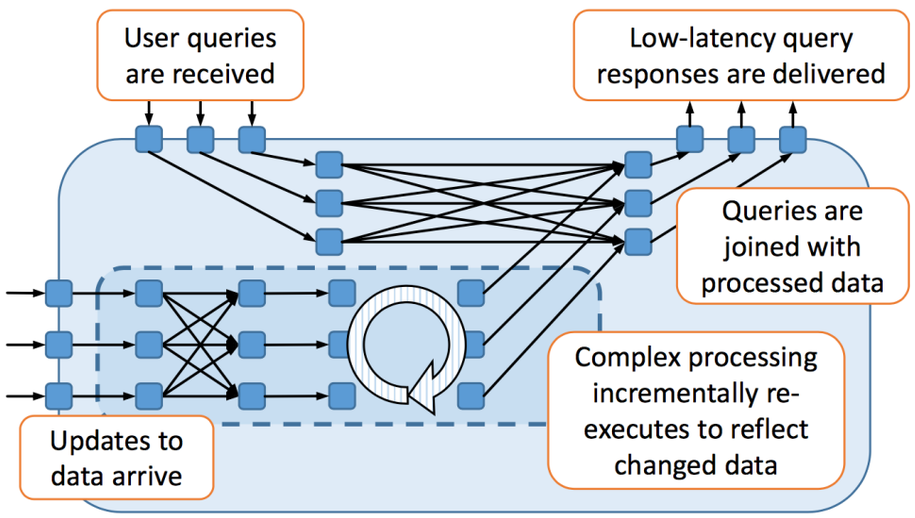
4. Performance

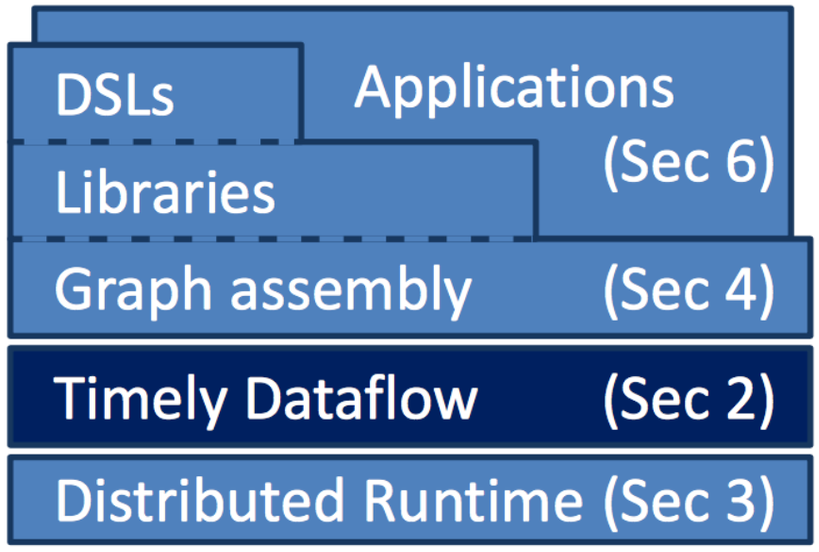
5. Real world Application

0. Introduction

Naiad is a distributed system for executing data parallel, cyclic data flow programs.
0. Batch Processor
1. Low latency Streaming processor
2. Iterative and incremental computation

Naiad => general-purpose system fulfils all these requirements to support high-level programming model.

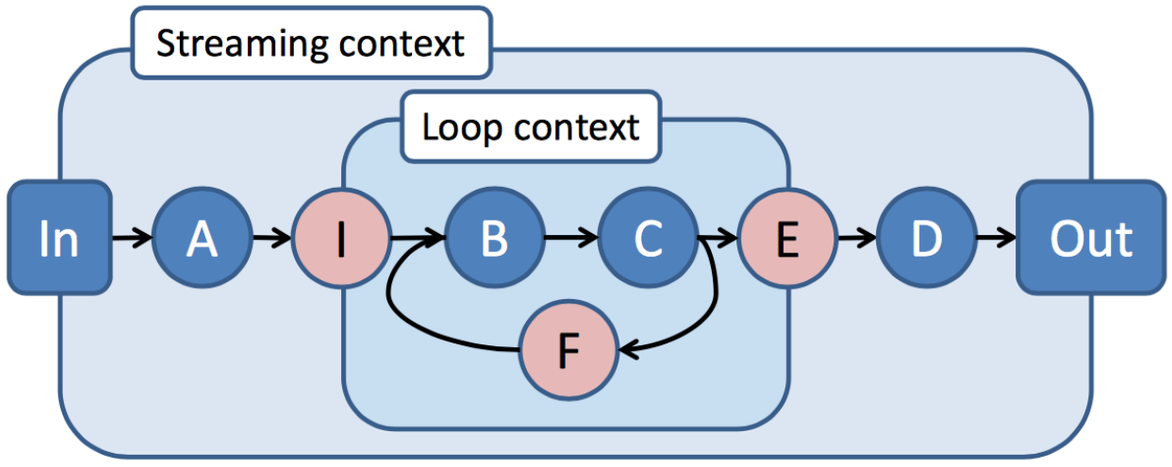




1. Timely Dataflow

Timely dataflow is a computational model based on ***directed*** graph.

Vertices: Send and receive logically timestamped messages.
Edges: Messages pass along the directed edges.



Input vertex: Receive messages from external producers.
Output vertex: Emit messages to external consumers.
Ingress vertex: Entrance for messages to a loop context.
Egress vertex: Exit of messages for messages to a loop context.
Feedback vertex: Messages pass through feedback to continue looping.

1. Timely Dataflow

$$\text{Timestamp} : (\overbrace{e \in \mathbb{N}}^{\text{epoch}}, \overbrace{\langle c_1, \ldots, c_k \rangle \in \mathbb{N}^k}^{\text{loop counters}})$$

Vertex	Input timestamp	Output timestamp
Ingress	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k, 0 \rangle)$
Egress	$(e, \langle c_1, \ldots, c_k, c_{k+1} \rangle)$	$(e, \langle c_1, \ldots, c_k \rangle)$
Feedback	$(e, \langle c_1, \ldots, c_k \rangle)$	$(e, \langle c_1, \ldots, c_k + 1 \rangle)$

t1 *<=* ***t2*** *iff* ***e1*** *<=* ***e2*** *and* ***vec(c1)*** *<=* ***vec(c2)***

(lexicographic ordering on integer sequences)

1. Timely Dataflow

v.OnRecv(e : Edge, m : Message, t : Timestamp)
v.OnNotify(t : Timestamp).

this.SendBy(e : Edge, m : Message, t : Timestamp)
this.NotifyAt(t : Timestamp).

ONRECV *and* ***ONNOTIFY*** : User implemented functions to handle a received message or doing something after being triggered by a specific time.

SENDBY and ***NOTIFYAT***: System provided functions to send a message to next vertex or trigger a vertex by a specific timestamp.

1. Timely Dataflow

```
class DistinctCount<S,T> : Vertex<T>
{
  Dictionary<T, Dictionary<S,int>> counts;
  void OnRecv(Edge e, S msg, T time)
  {
    if (!counts.ContainsKey(time)) {
      counts[time] = new Dictionary<S,int>();
      this.NotifyAt(time);
    }

    if (!counts[time].ContainsKey(msg)) {
      counts[time][msg] = 0;
      this.SendBy(output1, msg, time);
    }

    counts[time][msg]++;
  }

  void OnNotify(T time)
  {
    foreach (var pair in counts[time])
      this.SendBy(output2, pair, time);
    counts.Remove(time);
  }
}
```

Print distinguished messages to output1

Print messages counts to output2

OnNotify(t) is triggered iff no more OnRecv(e, msg, t') with t ≥ t'

when invoked with a timestamp t, the methods may only call SENDBY or NOTIFYAT with times t′ ≥ t.

1. Timely Dataflow

$$\text{Pointstamp} : (t \in \text{Timestamp}, \overbrace{l \in \text{Edge} \cup \text{Vertex}}^{\text{location}}).$$

Operation	Update
$v.\textsc{SendBy}(e,m,t)$	$\text{OC}[(t,e)] \leftarrow \text{OC}[(t,e)] + 1$
$v.\textsc{OnRecv}(e,m,t)$	$\text{OC}[(t,e)] \leftarrow \text{OC}[(t,e)] - 1$
$v.\textsc{NotifyAt}(t)$	$\text{OC}[(t,v)] \leftarrow \text{OC}[(t,v)] + 1$
$v.\textsc{OnNotify}(t)$	$\text{OC}[(t,v)] \leftarrow \text{OC}[(t,v)] - 1$

Reason about the ***impossibility of future messages*** bearing a given timestamp.
Set of timestamps at which future messages can occur is constrained by the current set of unprocessed *events.*

(t1, l1) could-result-in (t2, l2) $\psi = \langle l_1,\ldots,l_2 \rangle$ and $\psi\,(t_1) \leq t_2 \quad \Rightarrow \quad \Psi[l_1,l_2](t_1) \leq t_2$

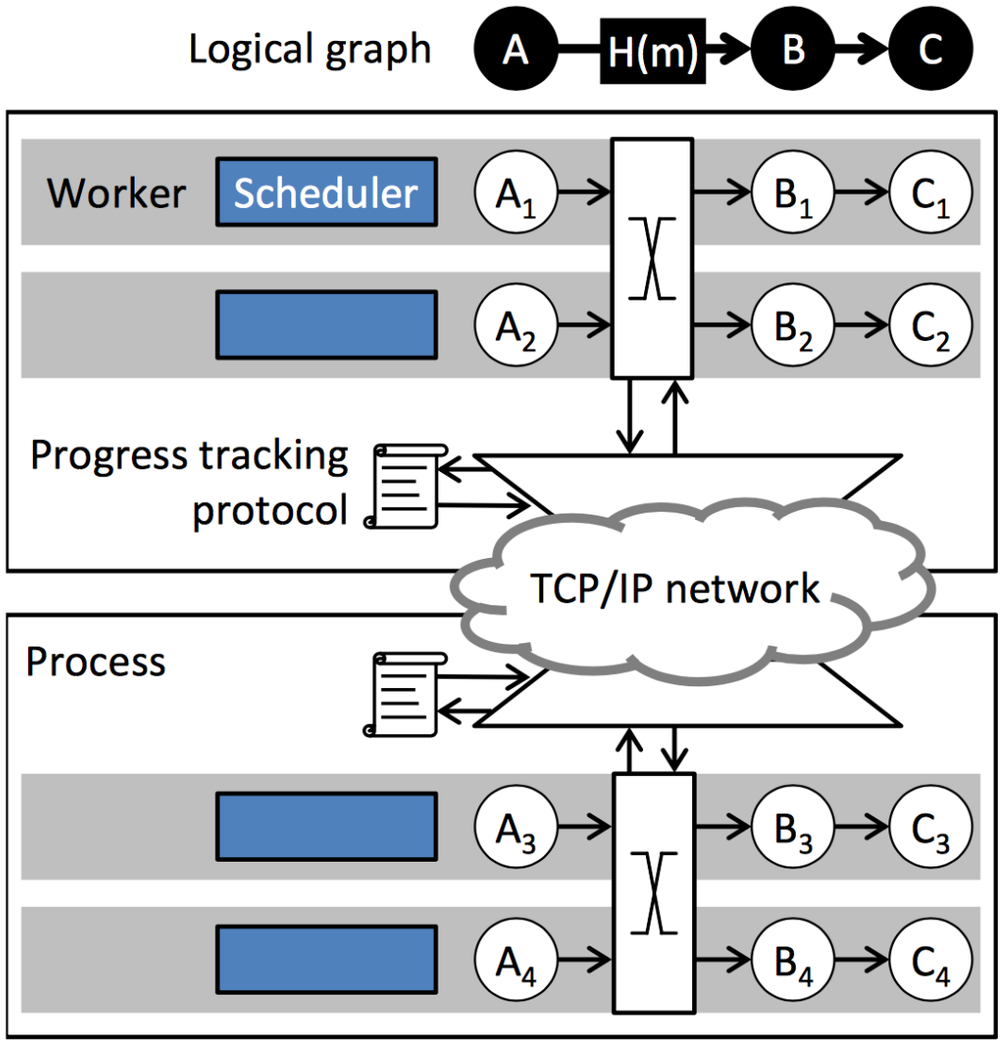
Maintains a set of ***active pointstamps(***at least one unprocessed)

occurrence count ***precursor count***

OC = 0 => Leave Active of P => Decrease PC that P could-result-in

PC = 0 => No more upstreams P that could-result-in => P is in frontier => Notification P at t

2.Implementation



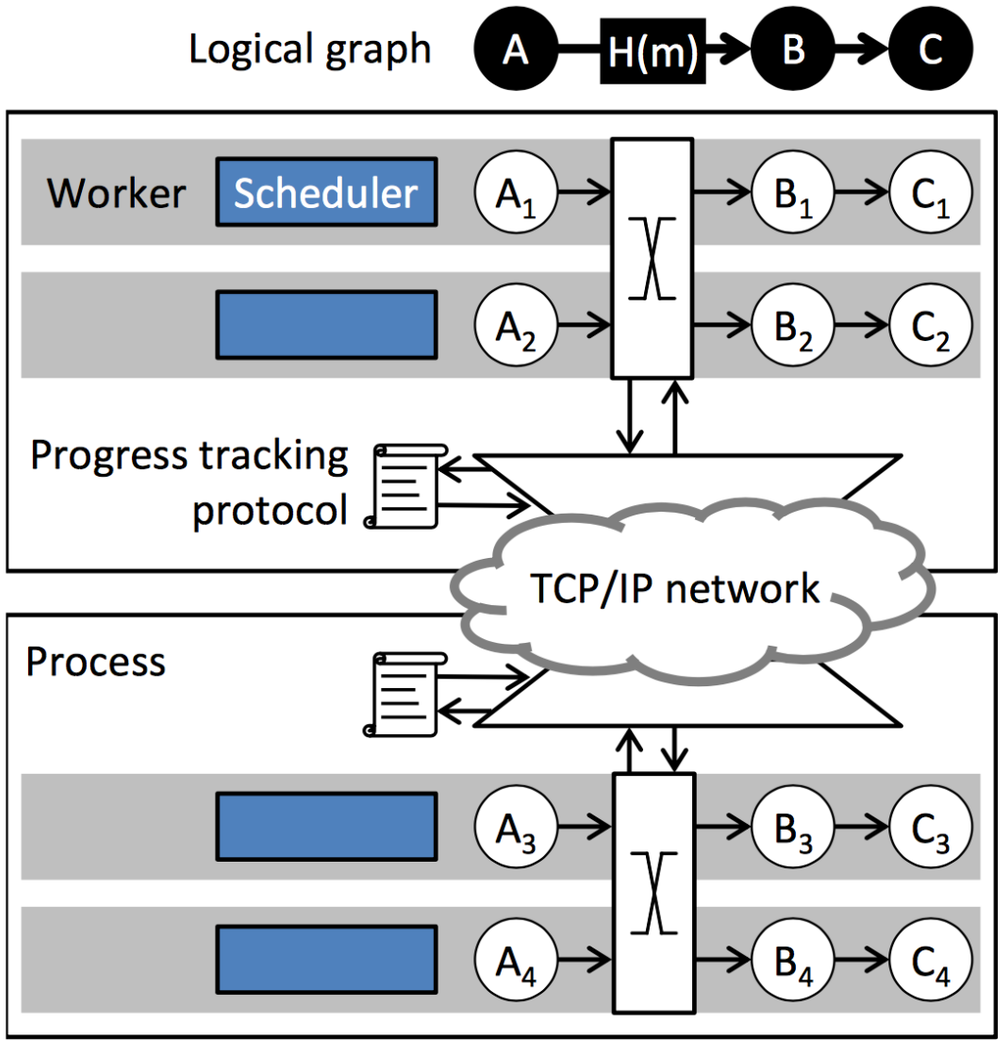
Data Parallelism

Logical graph of stages linked by typed connectors.

A logical vertex could be partitioned into several physical vertices allocating on different machines.

Connectors are able to let messages pass through different partitions on or not on the same machine.

2.Implementation



Workers

Worker is responsible for delivering messages and notifications to vertices in its partition of the timely dataflow graph.

Workers communicate with each other using shared queues and have no other shared state.

2.Implementation

Fault Tolerance

Checkpoint and ***Restore*** interface

During periodically checkpoints:
0. Pause worker and message delivery threads.
1. Flush unfinished message queues.
2. Do checkpoints.
3. Resume paused worker and message threads.
4. Flush checkpoints.

Recover failed vertex according to latest backup checkpoints using Restore method.

2.Implementation

Preventing micro-stragglers

Fact: Transient stalls at a single work will apparently affect on overall performance.

Package Loss => Use TCP.
Contention on concurrent data structure => Access through single thread.
Garbage collection =>Mark-and-Sweep GC and Engineer the system to trigger GC less frequently.

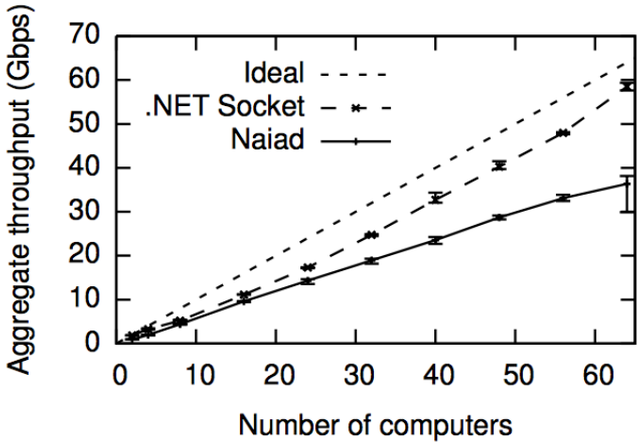
3.Write Programs on Naiad

```
// 1a. Define input stages for the dataflow.
var input = controller.NewInput<string>();

// 1b. Define the timely dataflow graph.
// Here, we use LINQ to implement MapReduce.
var result = input.SelectMany(y => map(y))
                  .GroupBy(y => key(y),
                     (k, vs) => reduce(k, vs))
```

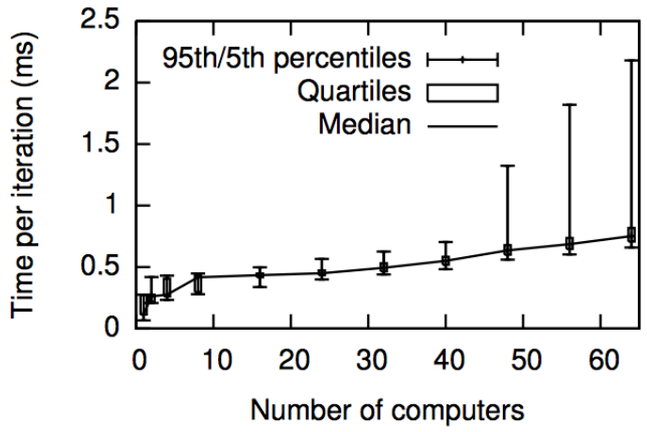
```
// 1c. Define output callbacks for each epoch
result.Subscribe(result => { ... });

// 2. Supply input data to the query.
input.OnNext(/* 1st epoch data */);
input.OnNext(/* 2nd epoch data */);
input.OnNext(/* 3rd epoch data */);
input.OnCompleted();
```

SQL
MapReduce
LINQ
Pregel's vertex program abstraction
PowerGraph's GAS abstraction

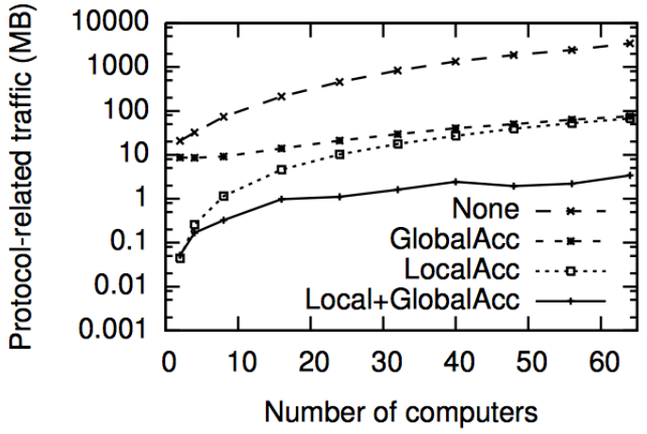
4.Evaluation



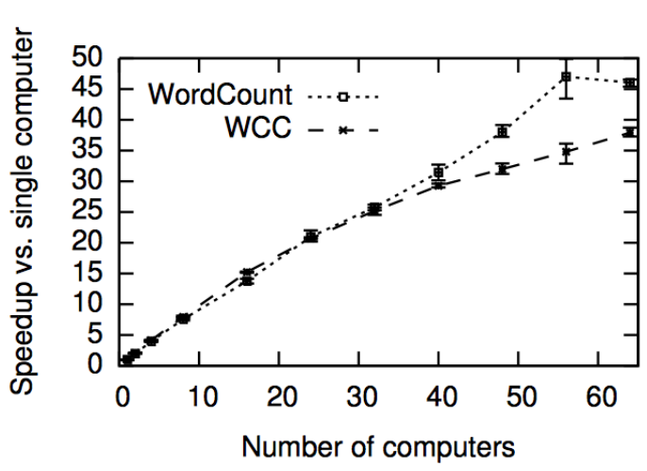
(a) All-to-all exchange throughput (§5.1)



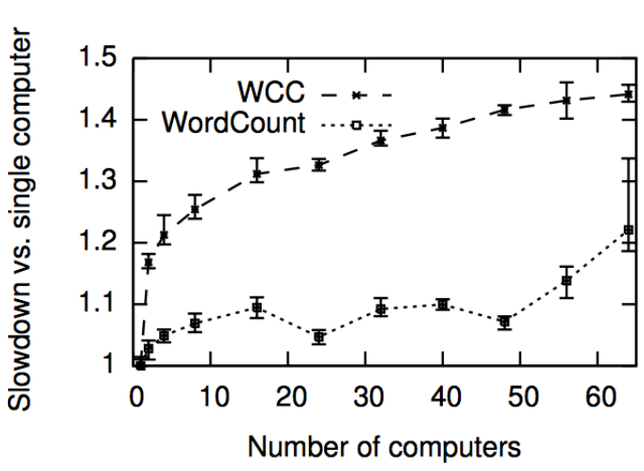
(b) Global barrier latency (§5.2)



(c) Progress tracking optimizations (§5.3)

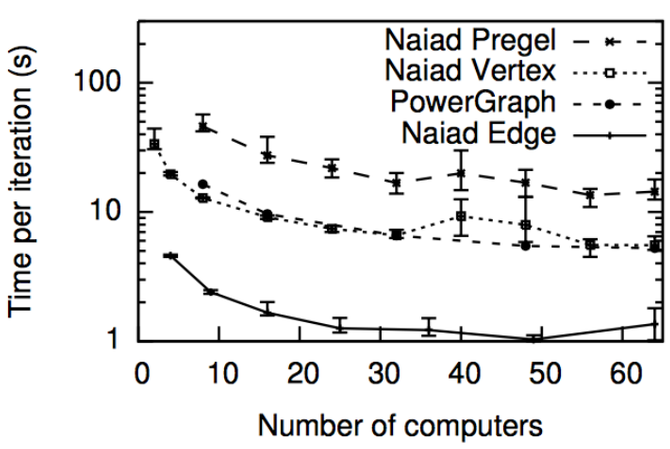


(d) Strong scaling (§5.4)

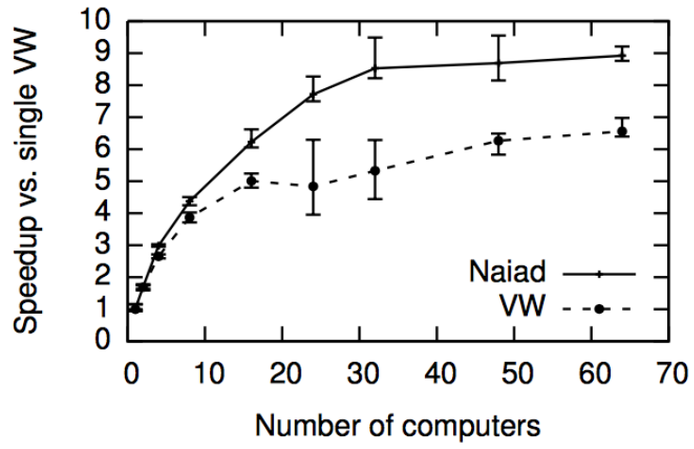


(e) Weak scaling (§5.4)

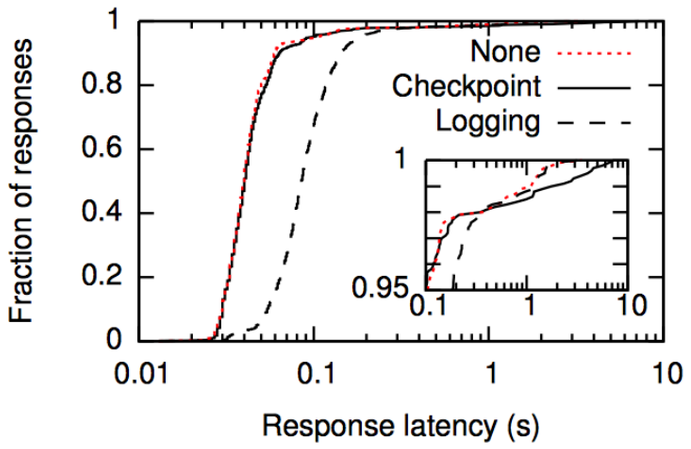
4.Real world Application



(a) PageRank on Twitter follower graph (§6.1)



(b) Logistic regression speedup (§6.2)



(c) k-Exposure response time (§6.3)

Algorithm	PDW	DryadLINQ	SHS	Naiad
PageRank	156,982	68,791	836,455	**4,656**
SCC	7,306	6,294	15,903	**729**
WCC	214,479	160,168	26,210	**268**
ASP	671,142	749,016	2,381,278	**1,131**

The general computational system Naiad is even more efficient than other specific aim system.