I/O Throttling and Coordination for MapReduce

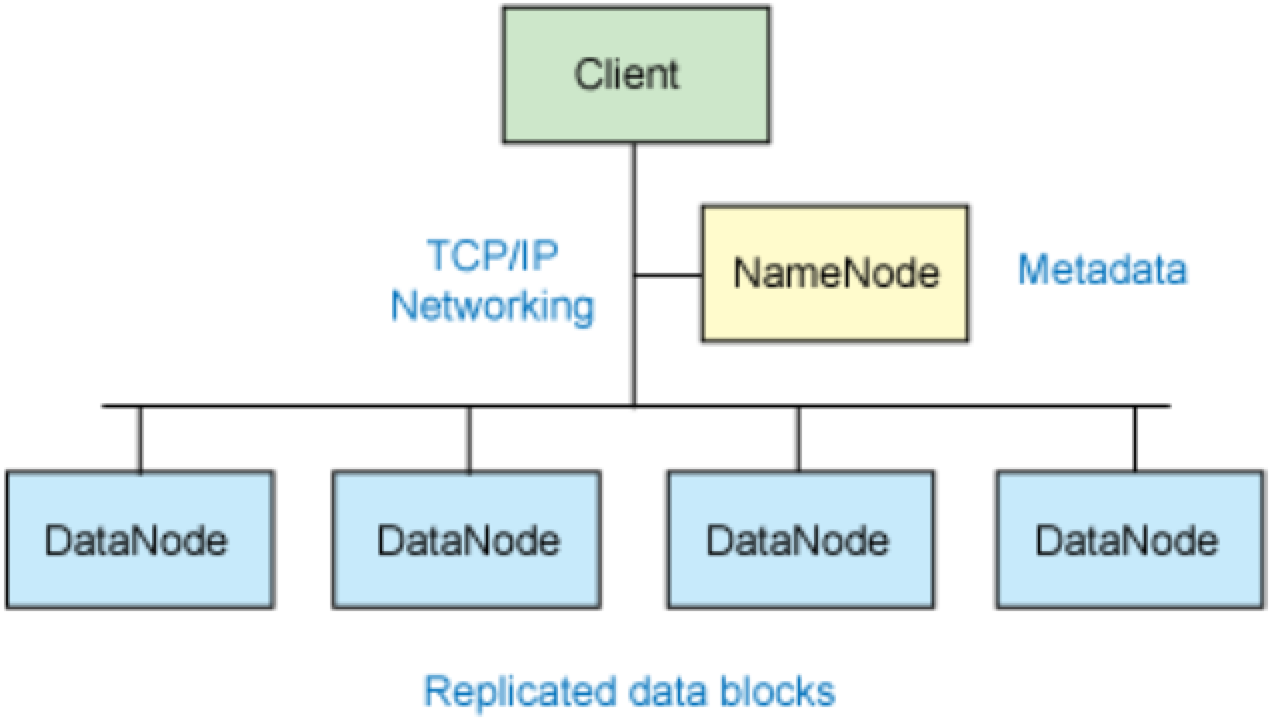
A Little Background on MapReduce

- Powerful framework for embarrassingly parallel problem
- Job = map tasks + reduce tasks
- Ease of programming and scaling up
- It is a programming model
- Also a distributed file system

A Little Background on MapReduce

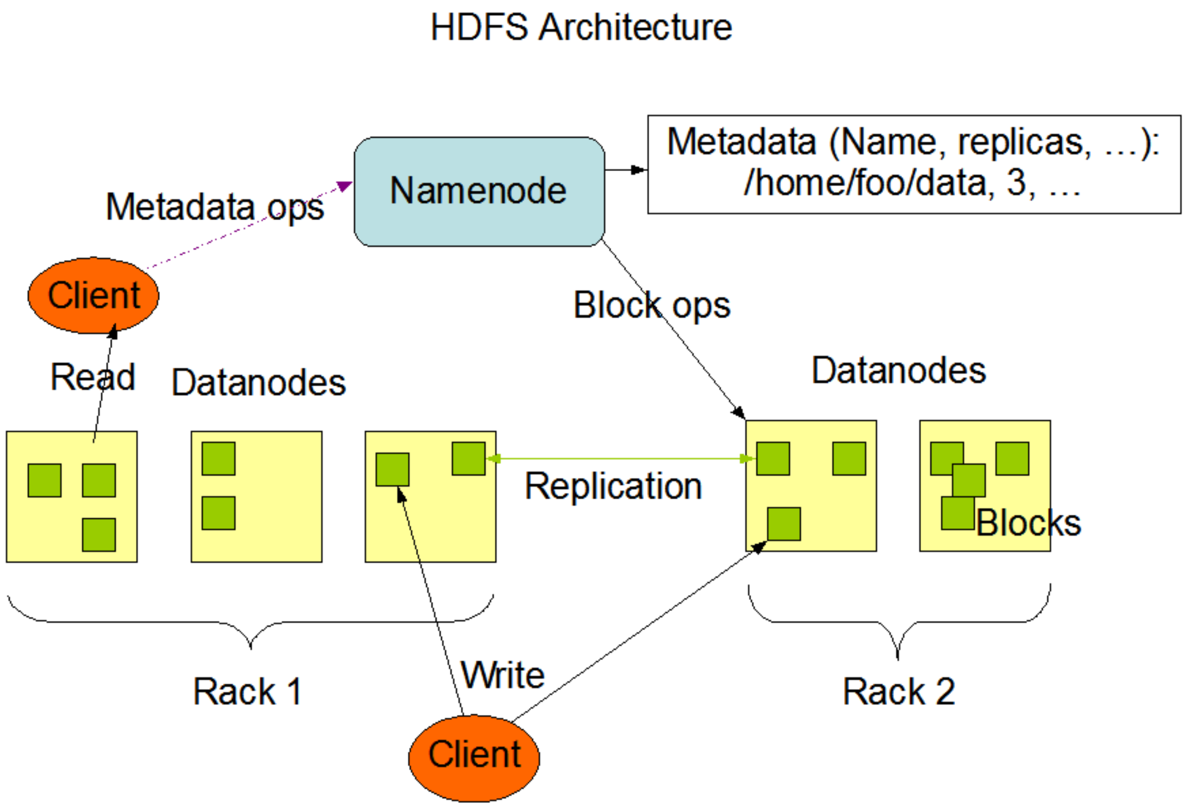
- Also a parallel file system, if let
  - application = User
  - I/O request = job
  - Strip = block

A Little Backgroud on Hadoop

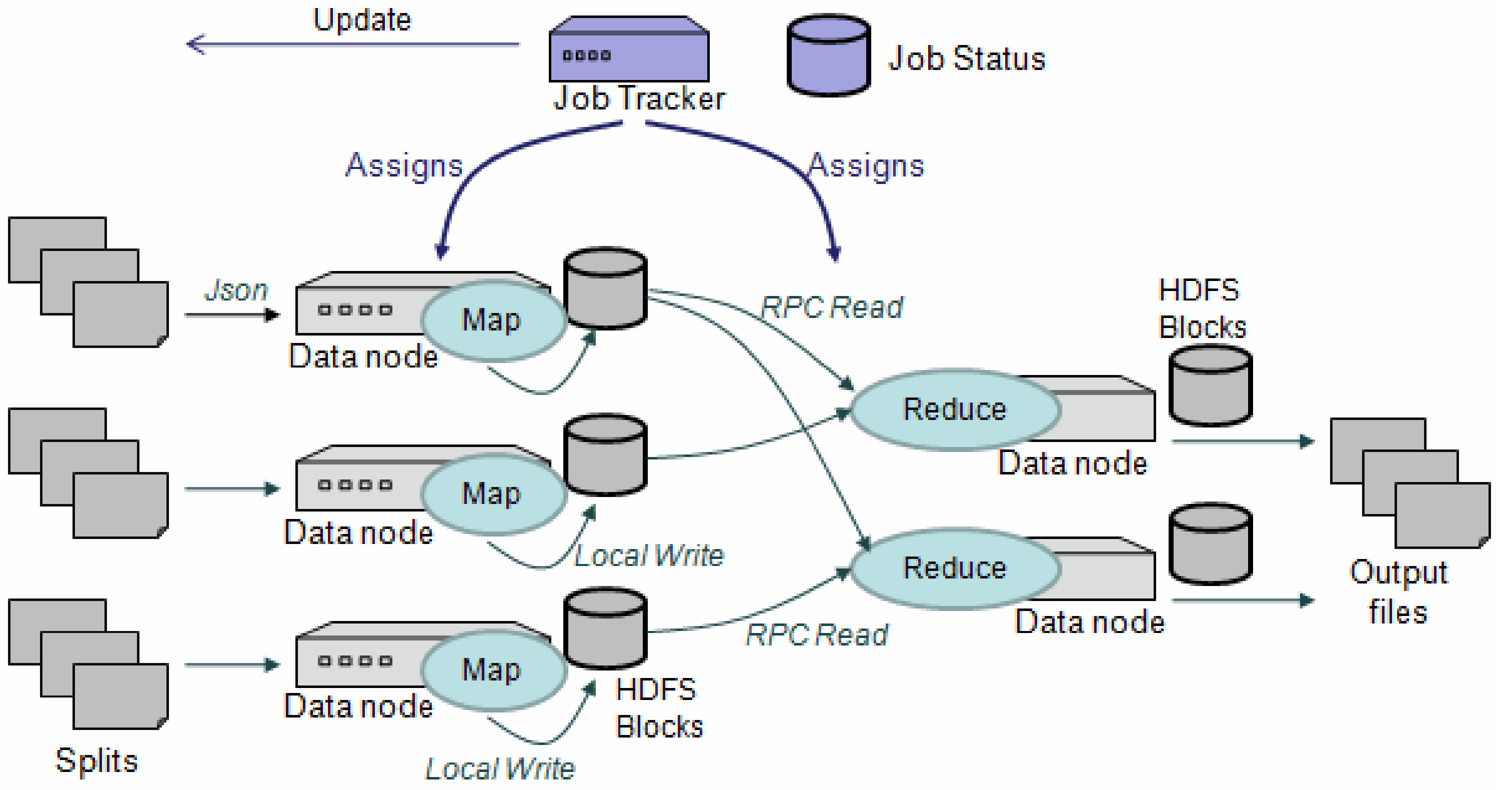


- Tasks access blocks through DataNode
- I/O accesses in Hadoop are blocking
  - Large write stream
  - Small read stream

A Little Backgroud on Hadoop



A Little Backgroud on Hadoop



A Little Backgroud on Hadoop

- Task Scheduler
  - Default: FIFO
    - Execute on job a time
  - FairScheduler
    - Multi-user
  - Capacity Scheduler

A Little Backgroud on Hadoop

- Very configurable
  - Hundreds of parameters
  - Companies works on selling configured Hadoop
    - cloundera

mapred.map.max.attempts	4	Expert: The maximum number of attempts per map task. In other words, framework will try to execute a map task these many number of times before giving up on it.
mapred.reduce.max.attempts	4	Expert: The maximum number of attempts per reduce task. In other words, framework will try to execute a reduce task these many number of times before giving up on it.
mapred.tasktracker.map.tasks.maximum	2	The maximum number of map tasks that will be run simultaneously by a task tracker.
mapred.tasktracker.reduce.tasks.maximum	2	The maximum number of reduce tasks that will be run simultaneously by a task tracker.
mapred.map.tasks.speculative.execution	true	If true, then multiple instances of some map tasks may be executed in parallel.
mapred.child.java.opts	-Xmx200m -Xms32m	Java opts for the task tracker child processes. The following symbol, if present, will be interpolated: @taskid@ is replaced by current TaskID. Any other occurrences of '@' will go unchanged. For example, to enable verbose gc logging to a file named for the taskid in /tmp and to set the heap maximum to be a gigabyte, pass a 'value' of: -Xmx1024m -verbose:gc -Xloggc:/tmp/@taskid@.gc The configuration variable mapred.child.ulimit can be used to control the maximum virtual memory of the child processes.
mapred.reduce.parallel.copies	5	The default number of parallel transfers run by reduce during the copy(shuffle) phase.
mapred.reduce.slowstart.completed.maps	0.05	Fraction of the number of maps in the job which should be complete before reduces are scheduled for the job

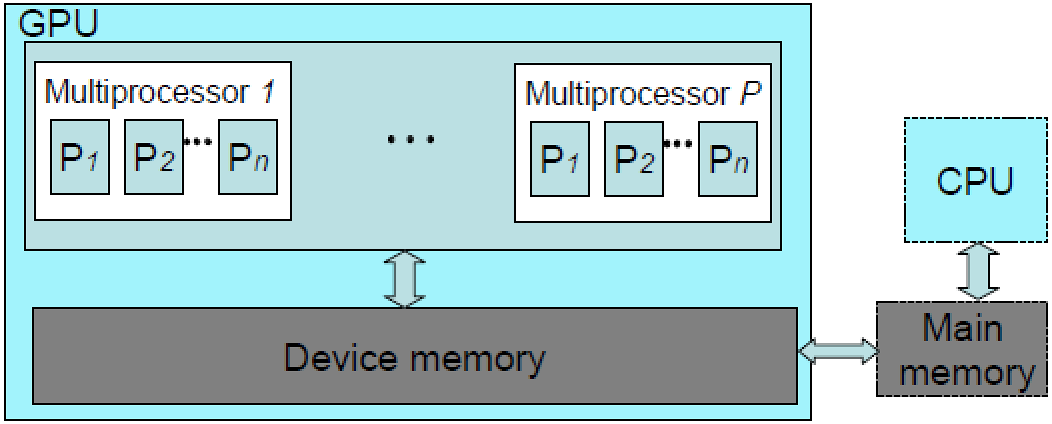
A Little Backgroud on Hadoop

ipc.client.timeout	60000	Defines the timeout for IPC calls in milliseconds.
mapred.task.timeout	600000	The number of milliseconds before a task will be terminated if it neither reads an input, writes an output, nor updates its status string.
dfs.datanode.socket.write.timeout	20000	The dfs Client waits for this much time for a socket write call to the datanode.
ipc.client.connection.maxidletime	10000	The maximum time in msec after which a client will bring down the connection to the server.

fs.inmemory.size.mb	200	Larger amount of memory allocated for the in-memory file-system used to merge map-outputs at the reduces.
io.sort.factor	100	More streams merged at once while sorting files.
io.sort.mb	200	Higher memory-limit while sorting data.
io.file.buffer.size	131072	Size of read/write buffer used in SequenceFiles.

Multi/many core

- PHOENIX - Multicore version
- MARS – GPU version

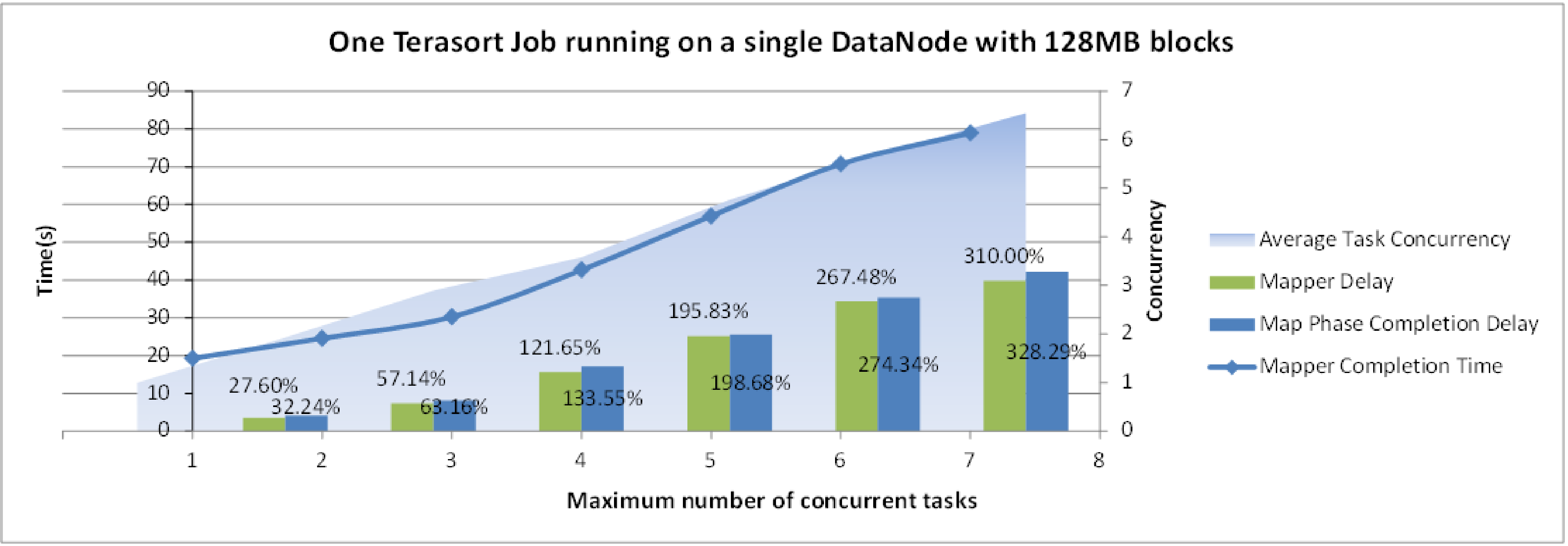


Motivation

Two Observations

Two Intuitions

Map Task Delay on a single Node



More concurrent tasks, more delay

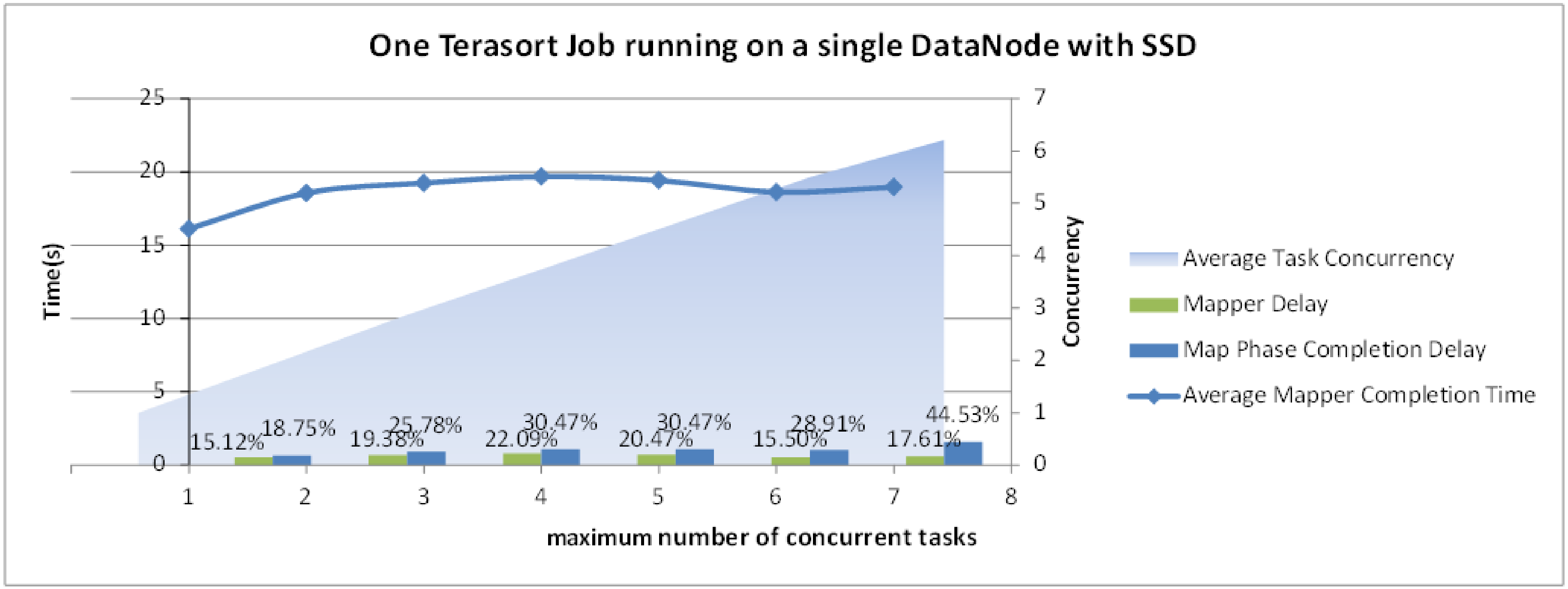
ONE OBSERVATION ON TASK LENGTH

we bet it is caused by I/O CONTETION!

?Need a proof?

ONE OBSERVATION ON **TASK LENGTH**

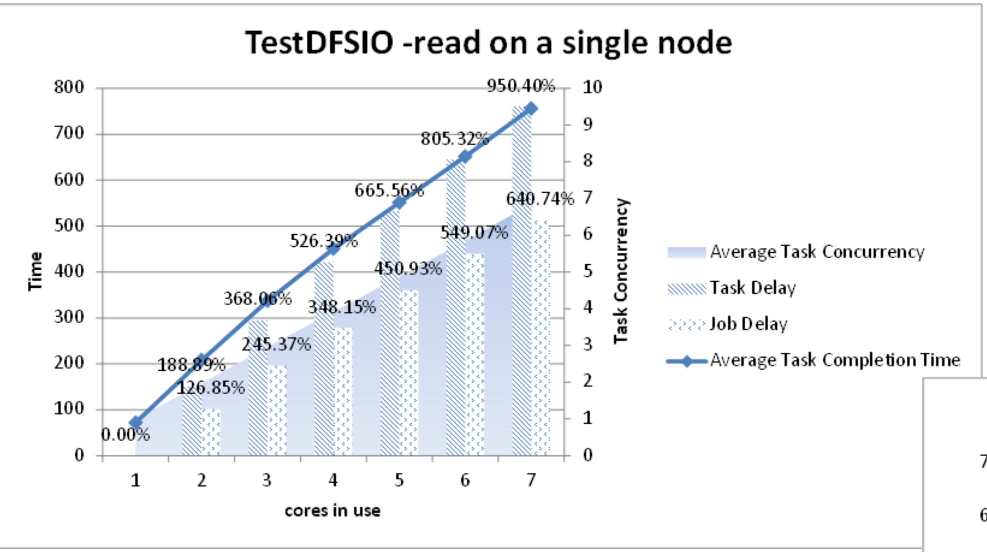
A quick ***PROOF***

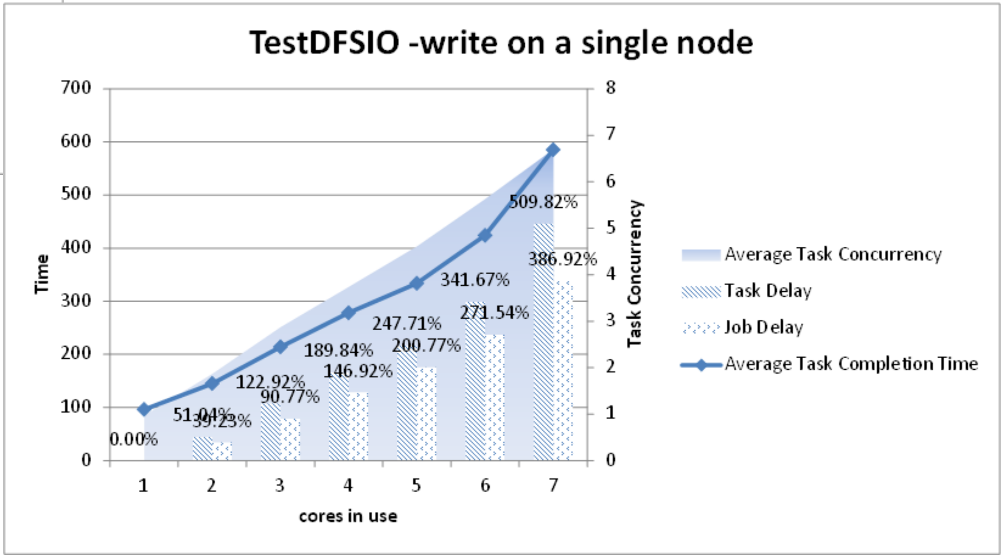


Much less delay after REMOVE the I/O bottleneck

ONE OBSERVATION ON TASK LENGTH

pure I/O Mapreduce Job





even WORSE

Motivation

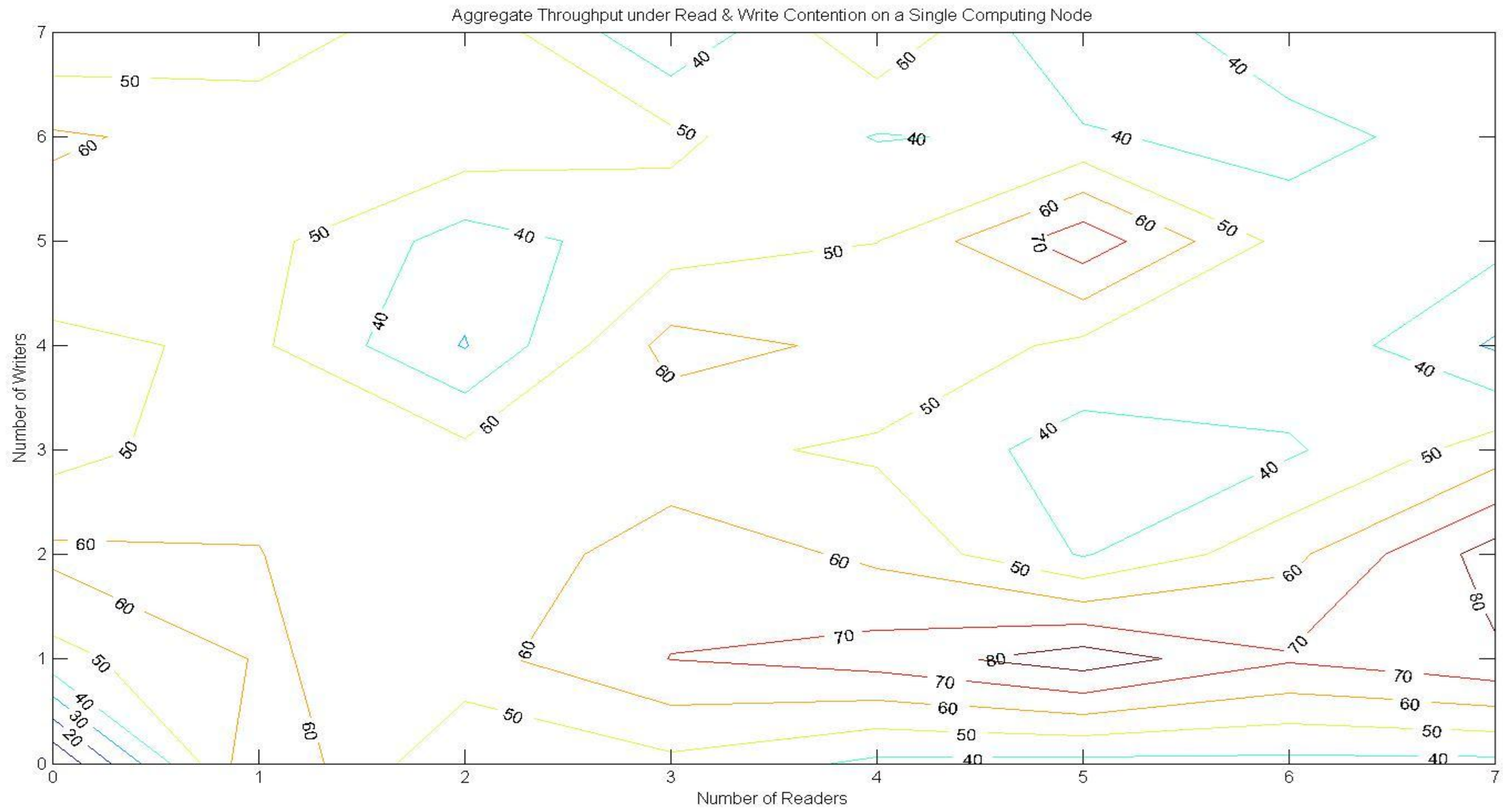
Two Observations

observation 1: I/O contention leads to general task delay

ONE OBSERVATION ON AGGREGATE THROUGHPUT

Throughput *FLUTUATES* as the increase of the number of concurrent tasks

Read & Write Conflicts with each other



Read drops to ONE TENTH of its original throughput

Motivation

Two Observations

observation 1: I/O contention leads to general task delay

observation 2: Concurrent I/O streams can be harmful

ONE INTUITION ON THROUGHPUT

The Throughput is PREDICTABLE knowing the number of concurrent read and write stream

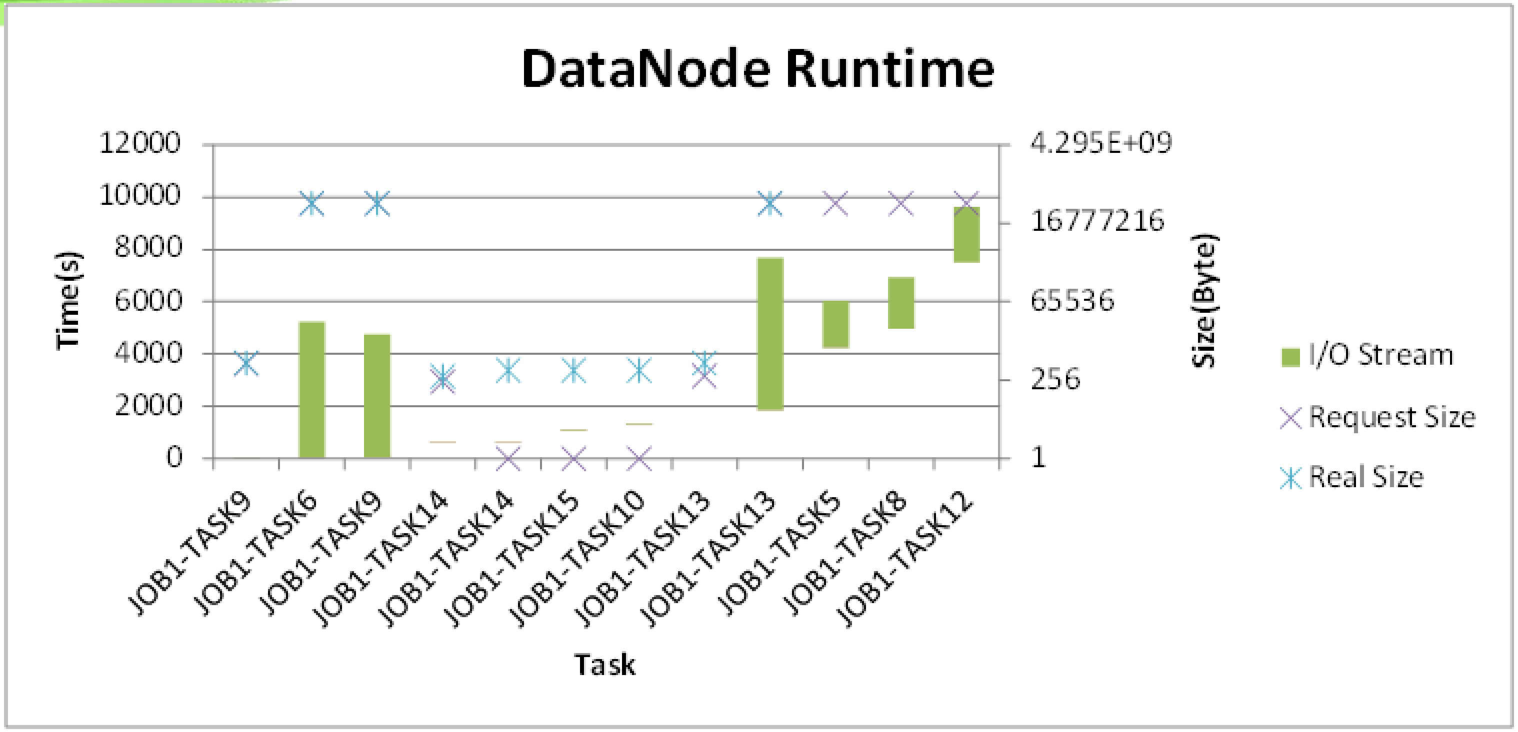
Reason1: block reserves locality

Reason2: MapReduce block is huge

Reason3: non-trivial I/O stream reads or writes entire block

Reason4: packet is flushed at the end of a write (->less cache influence)

ONE INTUITION ON THROUGHPUT



I/O stream is either quite large (128MB), or quite small (256B)

ONE INTUITION ON THROUGHPUT

The Throughput is PREDICTABLE knowing the number of concurrent read and write stream on a specified storage system

It indicates R-W concurrency control can MAXIMIZE system throughput

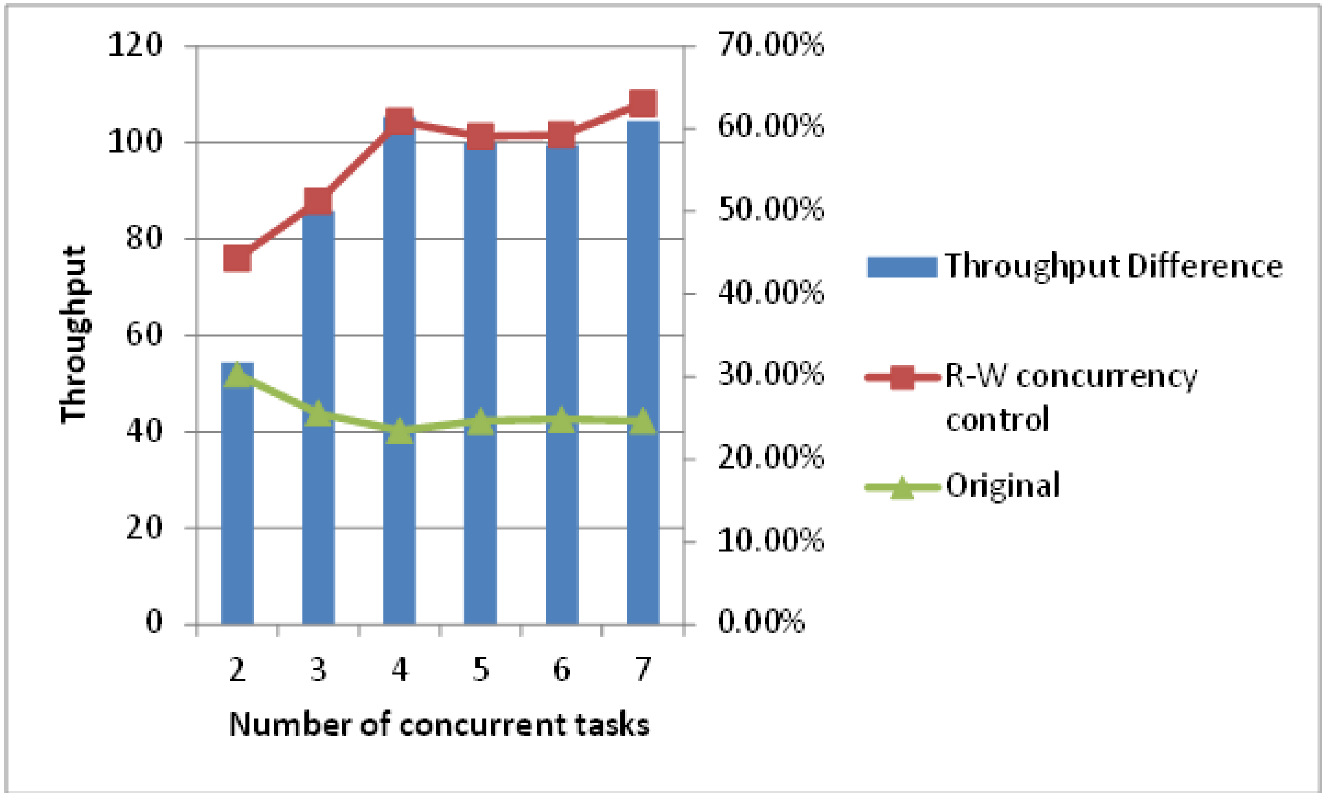
ONE INTUITION ON TASK DELAY

GENERAL Task Delay cause GENERAL Job Delay

Remove GENERAL on task delay, hence getting rid of GENREAL on job delay

Some Experimental Results

- ***Effect of I/O Throttling***



ONE INTUITION ON TASK DELAY

Notice I/O CONTENTION is the real problem

+

Take into account the job priority

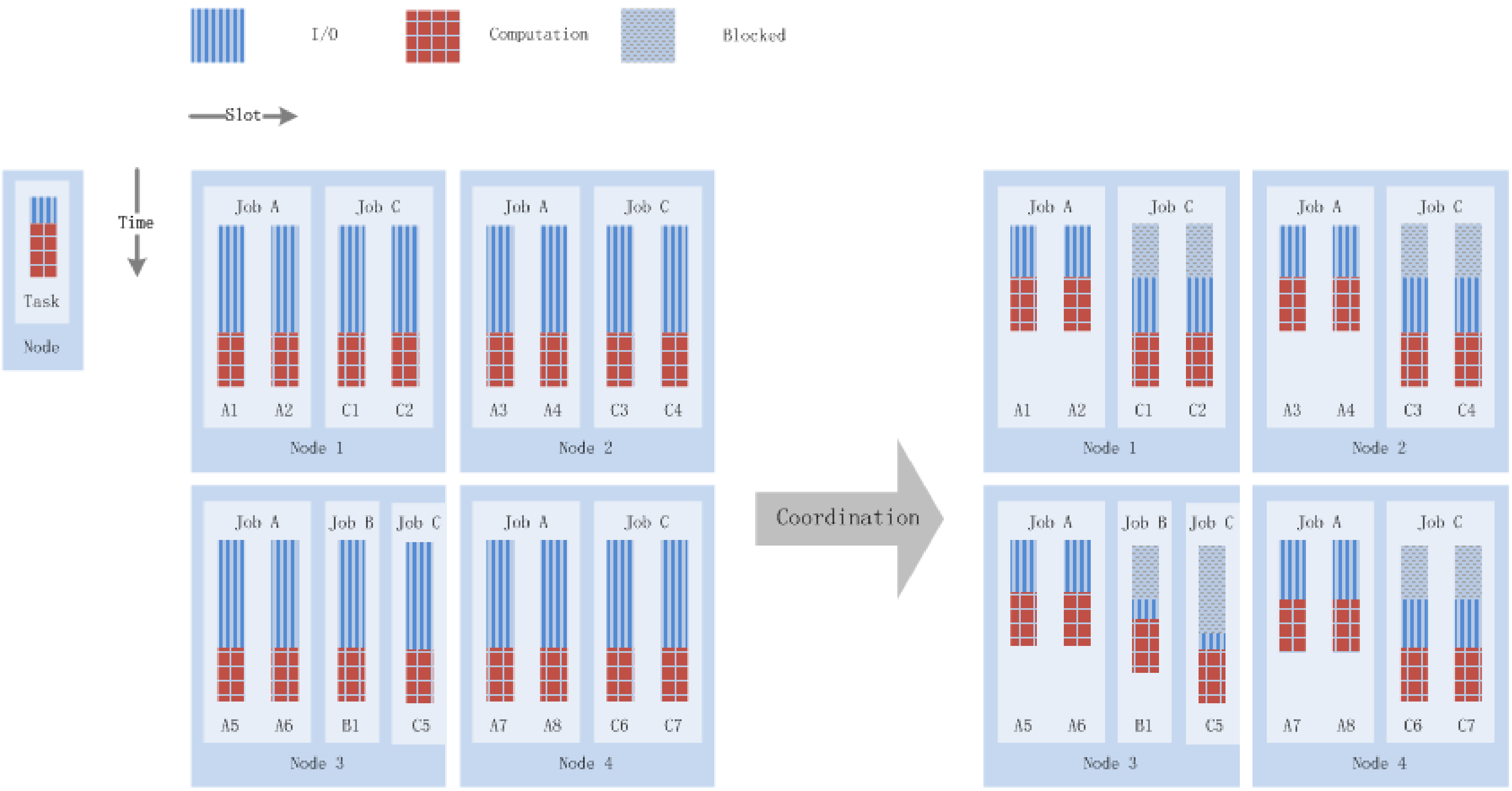
=

Give EXCLUSIVE I/O resource access to the tasks from HIGH PRIORITY jobs

=

I/O Coordination

An example assuming constant throughput



Motivation

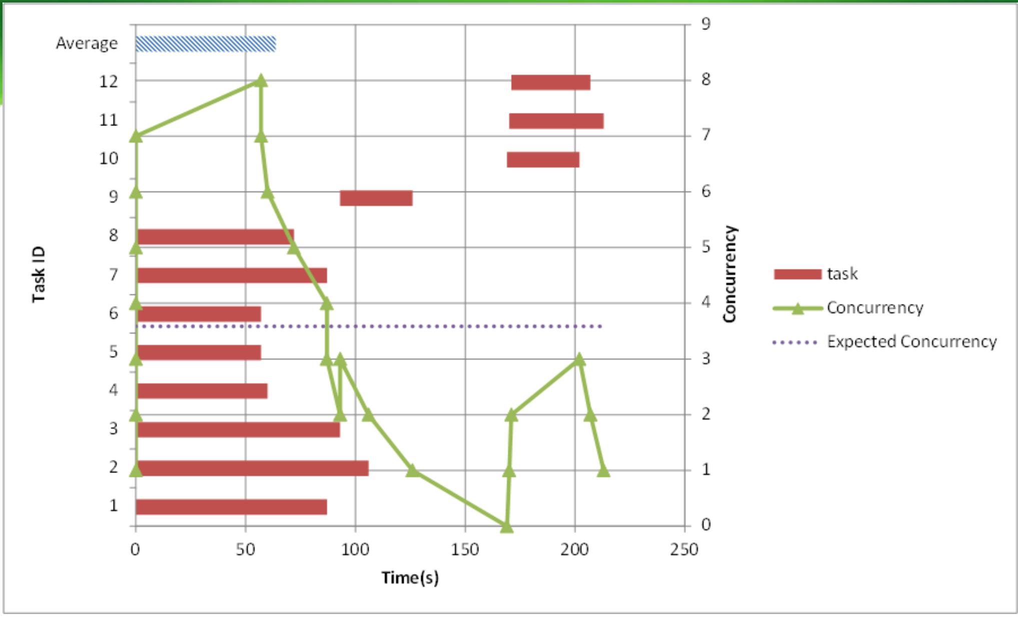
Two Observations

1. I/O contention leads to general task delay
2. Concurrent I/O streams can be harmful

Two Intuitions

1. R-W concurrency control can MAXIMIZE system throughput
2. Apply I/O coordination to REDUCE average job delay

Model



- **Some Definitions**

1. **System state S**
2. **Job A, task A1, A2, A3**
3. S_0 **is the contention free system**
4. $N_{task}(A)$ **is the number of tasks forming job A**
5. $\overline{C}_{task}^{job}(A, S)$ **is the expected number of concurrent tasks in job A running on system S**
6. $T_{task}(A_i, S)$ **indicates the completion time of task** A_i **on system** S**.**

Anatomy of MapReduce Job A's Completion Time

- **For Job A:**

$$T(A) = \overline{T}_{task}(A) \times \frac{N_{task}(A)}{\overline{C}_{task}^{job}(A)}$$

$$\overline{T}_{task}(A, S) = \frac{\sum_i T_{task}(A_i, S)}{N_{task}(A)}$$

we further dissect $T_{task}(A_j)$ into two parts, the non-I/O part and the I/O part

$$T_{task}(A_j, S_0) = T_{non-I/O}(A_j, S_0) + T_{I/O}(A_j, S_0)$$

Anatomy of MapReduce Job A's Completion Time

- Introduce I/O stretch factor:

$$ST_{I/O}^{task}(A_j, S) = \frac{T_{I/O}(A_j, S)}{T_{I/O}(A_j, S_0)}$$

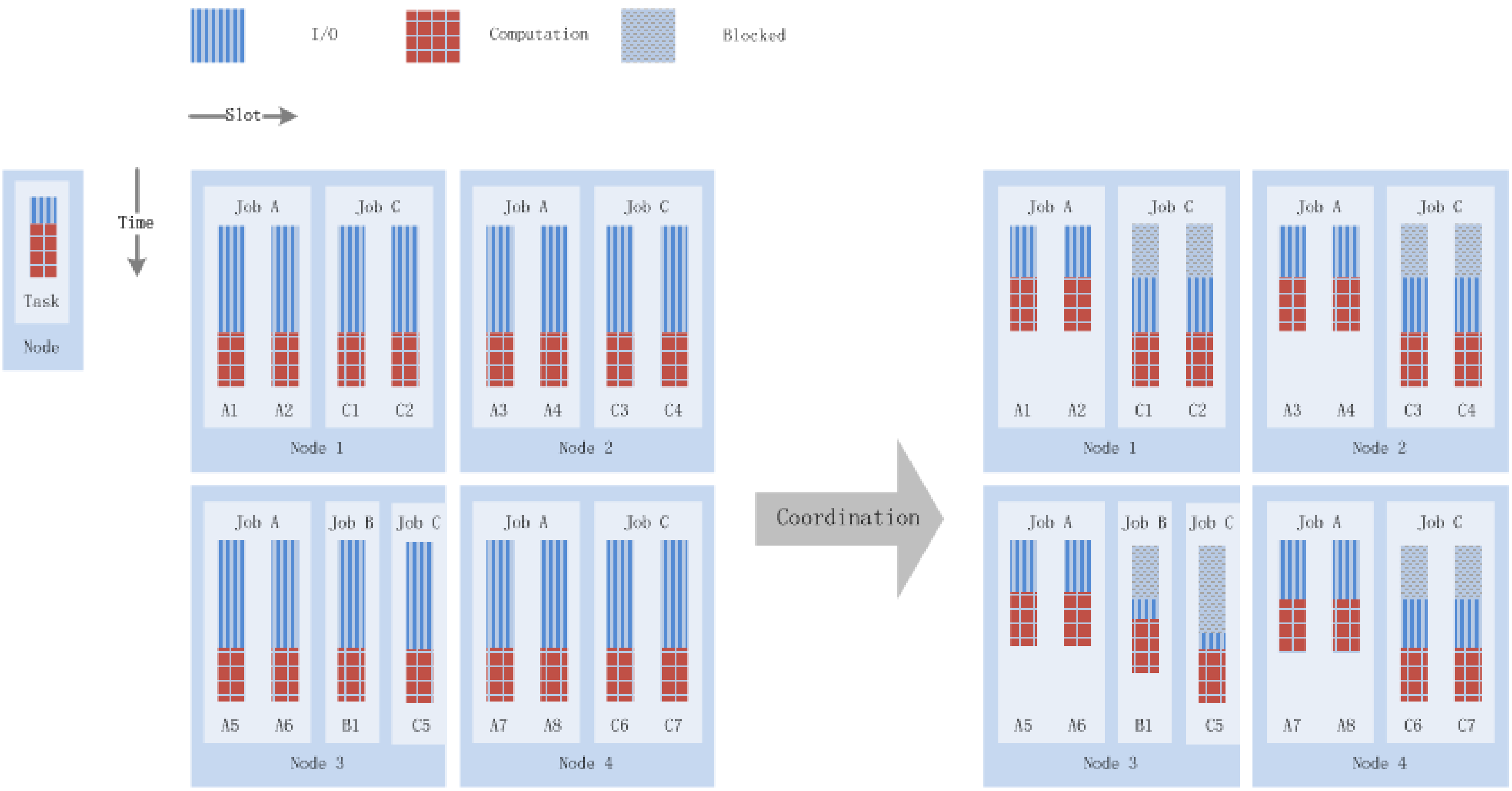
Then

$$T_{task}(A_j, S) = T_{non-I/O}(A_j, S) + T_{I/O}(A_j, S_0) \times ST_{I/O}^{task}(A_j, S)$$

How does I/O Coordination Work

- **It works by reducing STRETCH FACTOR for high priority tasks through R-W concurrency control**
  - **Still assume throughput is constant, and priority p(A) > p(B) > p(C)**
  - $ST_{I/O}^{task}(A_j, S)$ **= #expected read stream + #expected write stream**
  - **Before coordination:** $ST_{I/O}^{task}(\boldsymbol{t}, S) = \mathbf{4}$
  - **After coordination:** $ST_{I/O}^{task}(\boldsymbol{t}, S) = \mathbf{2}, \boldsymbol{t}\ \boldsymbol{in}\ \{\boldsymbol{A}\mathbf{1} - \boldsymbol{A}\mathbf{8}\}$ **and** $ST_{I/O}^{task}(\boldsymbol{B}\mathbf{1}, S) = \mathbf{3}$

How does I/O Coordination Work



How does I/O Coordination Work

- **By applying coordination, system state shifts from S_{old} to S_{new}**
- **Task I/O Portion** : $P_{I/O}(t) = T_{I/O}(t, S_0)/T(t, S_0)$
- $P_s(t, S_{old}, S_{new}) = P_{I/O}(t)(ST_{I/O}^{task}(t, S_{old}) - ST_{I/O}^{task}(t, S_{new}))$
- **Percentage of time that saved** $P_s(A, S_{old}, S_{new}) = \sum_{j \in A} P_s(A_j, S_{old}, S_{new}) / N_{task}(A)$ $= P_{I/O}(A) \sum_{j \in A} (ST_{I/O}^{task}(A_j, S_{old}) - ST_{I/O}^{task}(A_j, S_{new})) / N_{task}(A)$

Hardware Influence

- Faster Storage
- Faster CPU
- # cores on a computing node

Design

- R-W concurrency control
- Get a throughput matrix TP for the storage system
- $TP[i,j]$ = system throughput while i read stream and j write stream working at the same time
- Get a optimal R-W table RW
- $RW[i,j] = argmax(TP[s,t]), s \le i, t \le j$

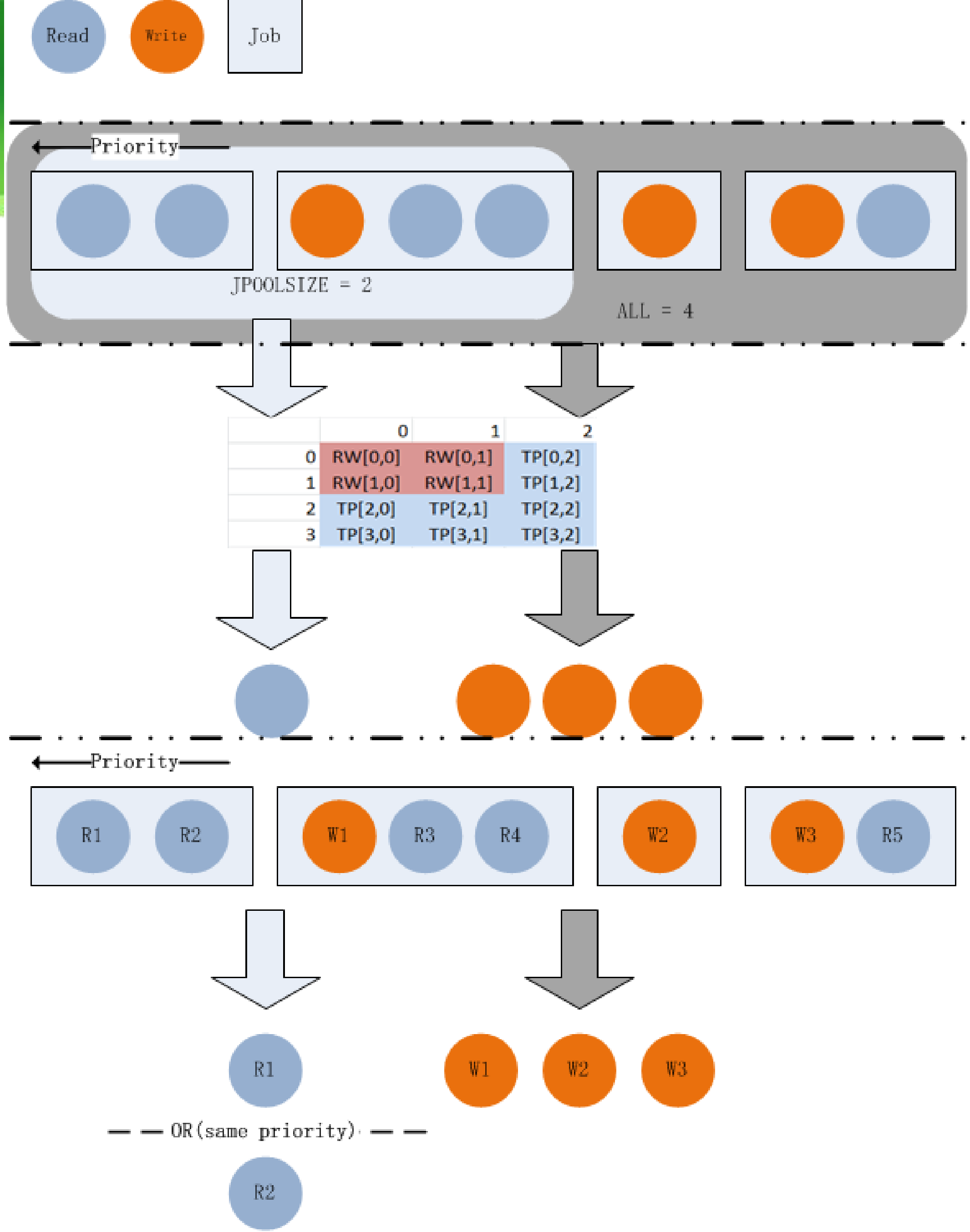
	0	1	2
0	0	69.16115	40.514
1	46.431	60.75237	56.45112
2	62.27	60.3143	50.31301
3	45.96	56.7809	52.43179

	0	1	2
0	(0,0)	(0,1)	40.514
1	(1,0)	(0,1)	56.45112
2	62.27	60.3143	50.31301
3	45.96	56.7809	52.43179

Design

- Ordered jobs by their priority
  - Job priority = (user-defined priority, submission time)
- Calculate the throughput G considering all the streams
- Calculate the throughput P in consideration of streams in the pool
- If (G-P)/G > MAXDIFF
  - Select G's solution
  - Else, P's solution
- Select the tasks base on their priority

Design



	0	1	2
0	RW[0,0]	RW[0,1]	TP[0,2]
1	RW[1,0]	RW[1,1]	TP[1,2]
2	TP[2,0]	TP[2,1]	TP[2,2]
3	TP[3,0]	TP[3,1]	TP[3,2]

I/O COORDINATION
order the jobs by their priority

R-W CONCURRENCY CONTORL
Choose proper number of read streams and write streams to maximize the throughput

I/O COORDINATION
select the tasks according to their priorities

Design

- ***Tradeoff: Throughput VS. Response Time***
  - ***MAXDIFF***
    - ***(G-P)/G > MAXDIFF***
    - ***Maximum throughput drop allowed***
  - ***JPOOLSIZE***
    - ***Small***
      - ***Much better response time for high priority job***
      - ***Miserable throughput***
      - ***Possible miserable average job response time***

Design(Alternative)

- ***Rule of Thumb***
  - ***Get no more than x read streams***
  - ***Get no more than y write streams***
- ***Reason***
  - ***Accuracy of the table***
  - ***Difficulty to capture the buffer state***
    - ***io.file.buffer.size***
      - ***Default: 4K***
      - ***Recommend: 128K***
      - ***Max: block size***
  - ***simple***
- ***Drawback***
  - ***Not optimal***

Design

- *Two techniques complements one another*
  - *R-W Concurrency Control*
    - *Input: a group of Read and Write Stream*
    - *Output: #Read Stream and #Write Stream that maximize the throuput*
    - *How to select Read and Write stream?*
  - *I/O Coordination*
    - *Input: a group of streams*
    - *Output: a group of streams that have the highest priority*
    - *Which stream to be selected?*

Implementation

- ***DataXeiver.java***
  - ***readBlock()***
  - ***writeBlock()***

```
readBlock(...) {
    ...
    blockSender = new BlockSender(...);
    blockSender.coOn = true;
    ...
}

writeBlock(...) {
    ...
    blockReceiver = new BlockReceiver(...);
    blockReceiver.coOn = true;
    ...
}
```

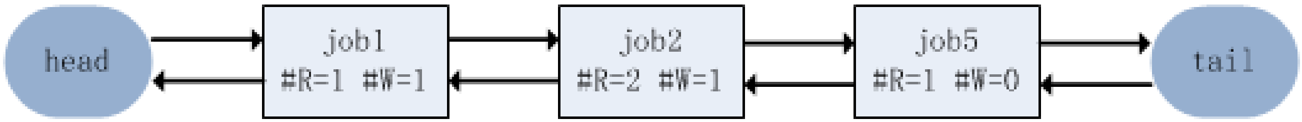
```
sendBlock(...) {
    ...
    while(endOffset > offset) {
        if(coOn)
            coordination();
        len = sendChunks(...);
        offset += len;
    }
    ...
}

receiveBlock(...) {
    ...
    while(receivePacket()>0) {
        if(coOn)
            coordination();
    }
    ...
}
```

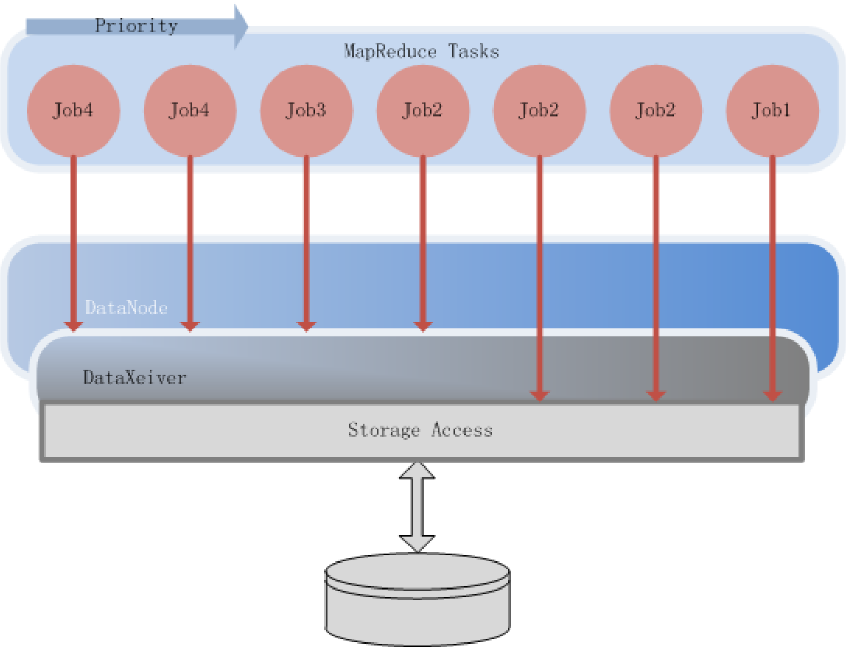
Implementation

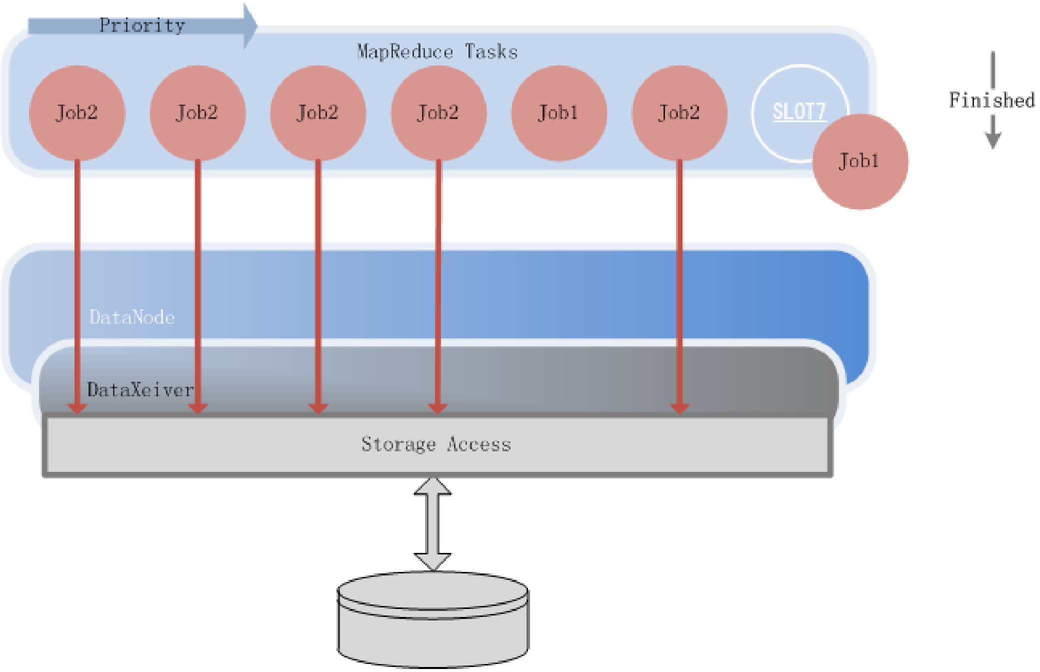
- *BlockSender.java*
  - *SendBlock()*
- *BlockReceiver.java*
  - *ReceiveBlock()*
- *DataOrchestrator.java*
  - *Structure for synchronization*
  - *Block unauthorized I/O stream*
  - *Re-check the blocking condition for every chunk(read) and packet(write)*
    - *able to suspend Stream in the middle of serving*

```java
while(!isMyTurn2Read(id)) {
	synchronized(this){
		try {
		this.wait(TIMEOUT);
		} catch(InterruptedException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
		}
	}
}
```

Implementation

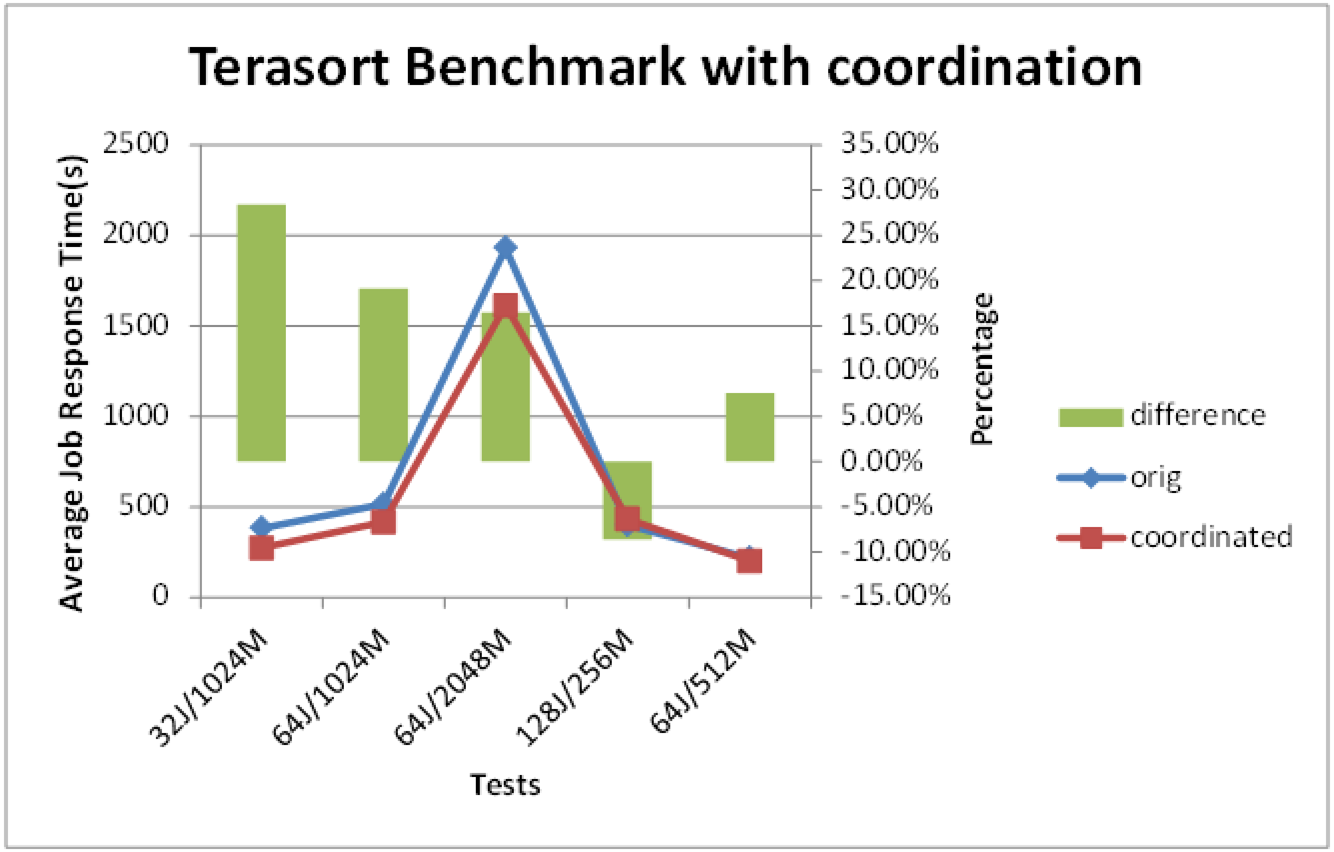






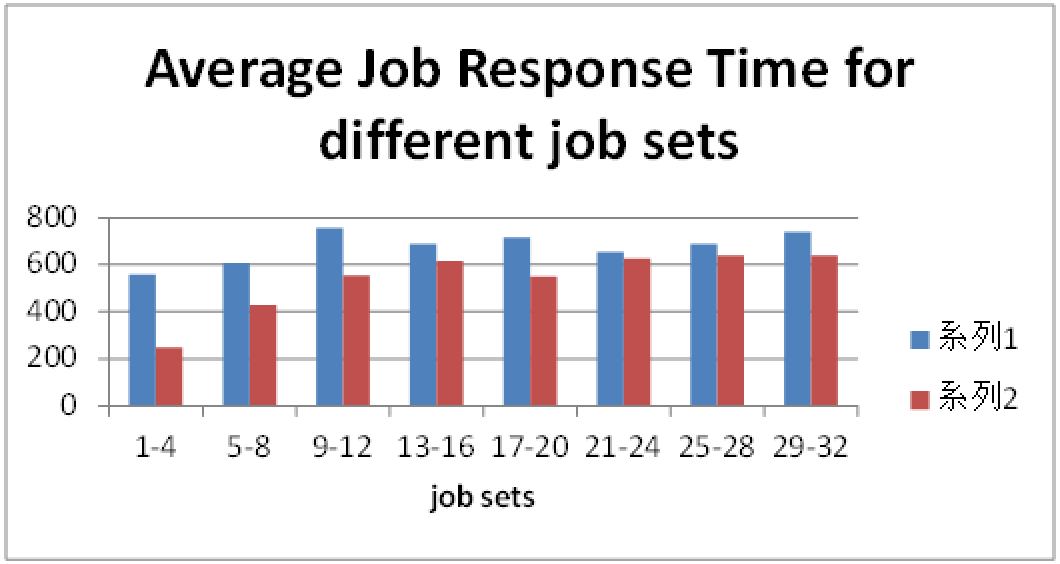
Some Experimental Results

- ***Configuration: poolsize=2***



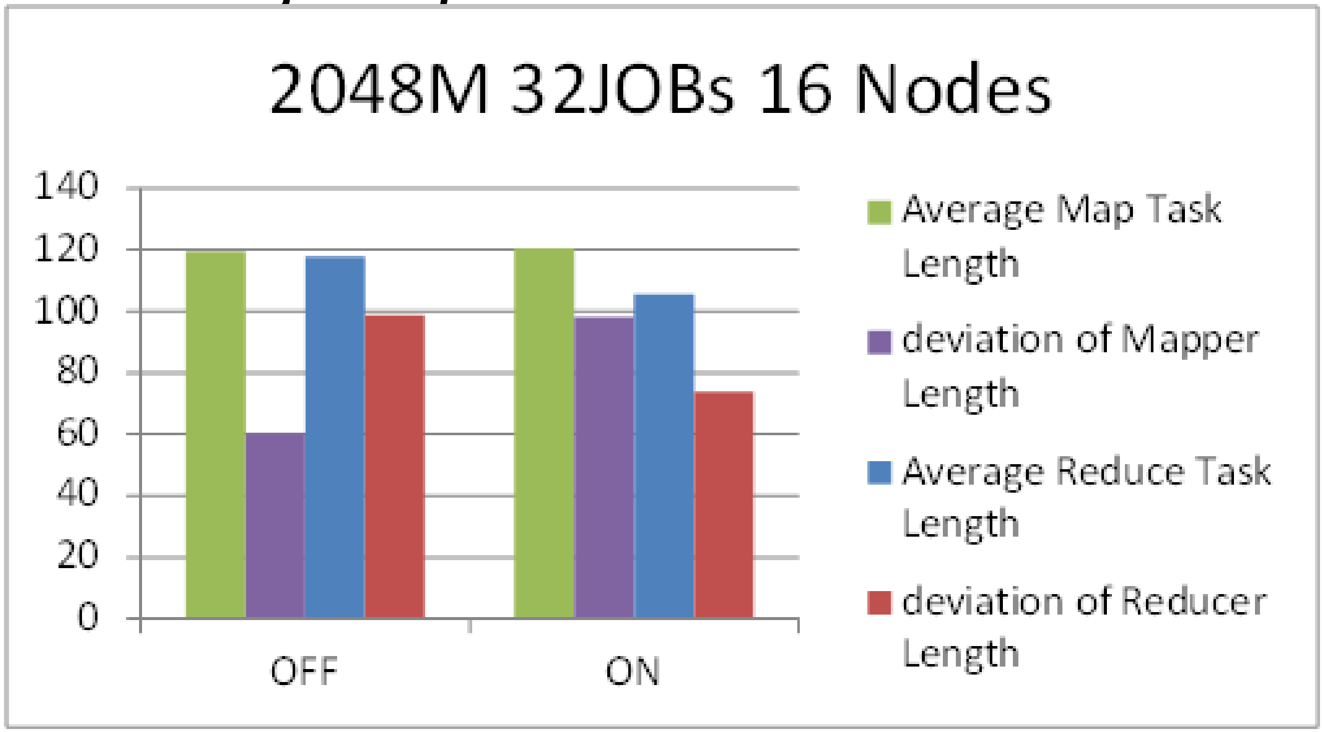
Some Experimental Results

- ***When job is small: 512M/128M = 4 blocks***
  - ***effective for job response time***
  - ***Sensitive to priorities***



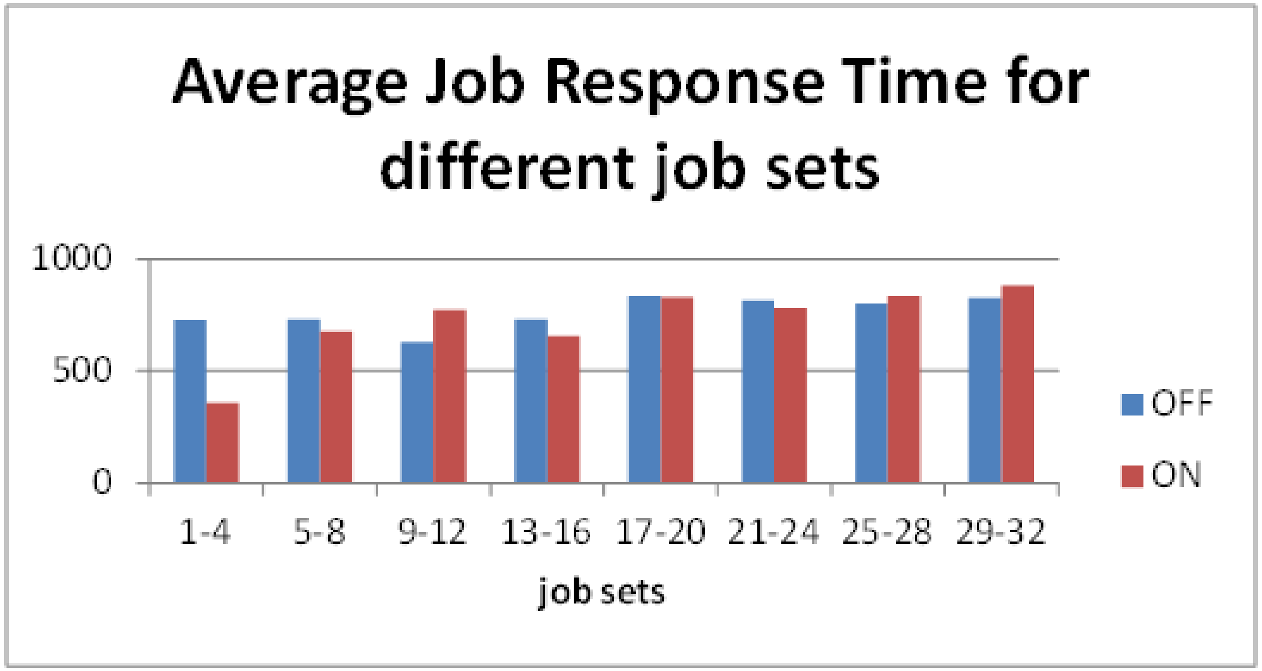
Some Experimental Results

- ***When job is large: 2048M/128M = 16 blocks***
  - ***Less effective for job response time***

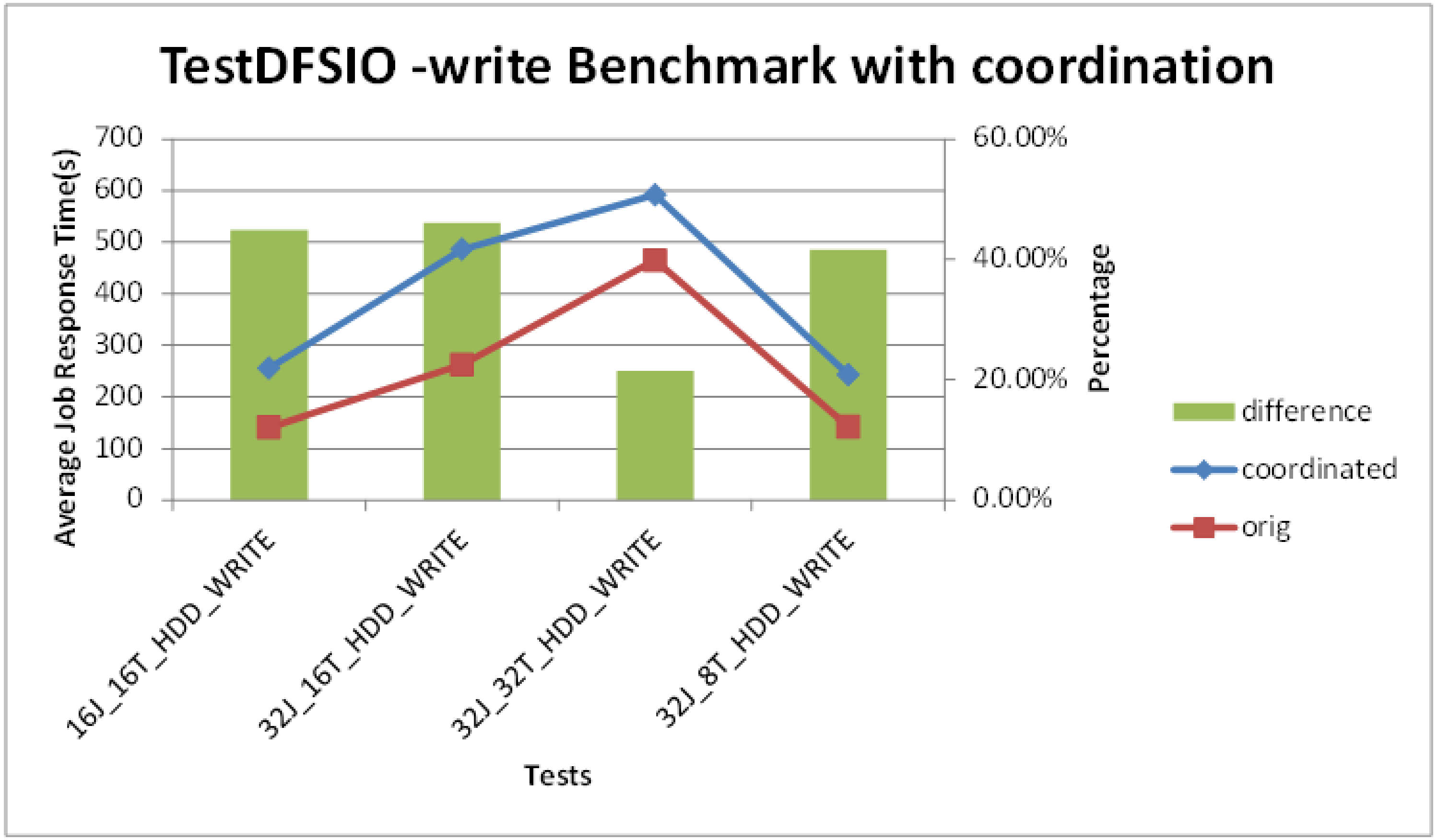


Some Experimental Results

- ***When job is large: 2048M/128M = 16 blocks***
  - ***Still effective in regard to QoS***



Some Experimental Results



Future Work

- ***Make this work an official patch for Hadoop***
- ***Other shared resources***
  - ***Network contention (Network I/O coordination)***
  - ***Bus, Cache, Memory Controller in many core***
- ***Virtual Environment***
- ***More QoS Oriented***

Thanks