CS423 – Operating Systems: MP4

Anirudh Jayakumar (ajayaku2) Debjit Pal (dpal2) Neha Chaube (nchaub2) and Divya Balakrishna (dbalakr2)

# Flow:

The process is started on each node with the node ID as command-line argument. The same configuration file is provided to both the nodes. Process 0 acts the master. It is responsible for creating the workload as well as detecting completion of the processing. The master creates the jobs and sends half of the jobs to the remote. Then the nodes start processing the jobs. The adaptor tries to load balance the system based on the algorithm configured. The master then periodically checks for completion of processing. Once it detects the end, it initiates a transfer of completed jobs from the remote to the master. Once it receives all the completed jobs, the jobs are checks for integrity to see if they have been processed correctly.

# Design:

## Adaptor:

The adaptor is the main component that controls the load balancing activities in the system. The APIs that Adaptor exposes are given below.

|  |  |
| --- | --- |
| **int** **Initialize**(CStateManager \*, CTransferManager \*,configInfo \*); | Initializes the thread that pools for load balancing |
| **void** **StopThread**(); | Stops the thread |

## CommServer

CommServer is composed of the RPC server. RPC server is implemented in Apache thrift. The commserver takes the IP and Port from the config xml. The APIs exposed by CommServer is given below.

|  |  |
| --- | --- |
| **void** **Init**(configInfo \*, CTransferManager \*,CStateManager\*); | Initializes the thread that listens to the port configured in the xml. |
| **int** **WaitServer**(); | Waits for the server to join when it is terminated |

## CommProxy

CommProxy implements the client to the RPC server. All communication to the remote node is done through the proxy.

|  |  |
| --- | --- |
| **int** **Initialize**(configInfo \*config); | Initializes the proxy with the configuration needed to connect to the server |
| **int** **UnInitialize**(); | Uninitializes and closes the server |
| **int** **SendJobsToRemote**(std::vector<CJob\*> &vJobs); | Proxy method to send jobs to remote node during load balancing |
| **int** **RequestJobsFromRemote**(**int** nJobs); | Proxy method to request for jobs during load balancing |
| **int** **RequestStateFromRemote**(); | Proxy method to request state from the remote node |
| **int** **SendStateToRemote**(State \*state); | Proxy method to send state to remote node |
| **int** **RequestCompletedJobsFromRemote**(); | Proxy method to request completed jobs from remote |
| **int** **SendCompletedJobsToRemote**(std::vector<CJob\*> &vJobs); | Proxy to send completed jobs to remote. Called during aggregation |

## HWMonitor

Hardware monitor periodically monitors the state of the system and stores in the local state of the monitor object to be retrieved by the state manager. The API exposed by this component is given below.

|  |  |
| --- | --- |
| **int** **Initialize**(configInfo \*info); | Initializes the hardware monitor and starts a thread and monitor periodically based on the configurations |
| **double** **GetCPUUtil**(); | Get method to access the cpu utilization |
| **float** **GetThrottlingValue**(); | Get method to access the throttling |
| **int** **SetCPUUtil**(); | Set the CPU utilization |
| **int** **SetThrottlingValue**(); | Set throttling value |

## JobQueue

This component handles the job queue, both pending and completed. APIs include methods to access the queues. We use a STL deque for the pending job queue so that we can treat the data structure as a queue as well we can access the end of the list to slice jobs to transfer to other node for load balancing. APIs are given below.

|  |  |
| --- | --- |
| CJob\* **GetNextJob**(); | Gets the next job from the queue |
| JobVec **SliceChunkFromQueue**(**int** nJobs); | Gets a chunk of jobs from the end of the queue |
| **void** **IntegrityCheck**(); | Checks if the job is processed completely |
| JobVec **GetCompletedJobs**(); | Get the list of completed jobs |
| **int** **AddJobsToQueue**(JobVec &vJobs); | Add jobs to the pending queue |
| **int** **AddCompletedJob**(CJob \*job); | Add job to the completed queue |

|  |  |
| --- | --- |
| **Int** **AddCompletedJobsToQueue** (JobVec &vJobs); | Add a list of jobs to completed queue. Used during aggregation at the master |
| **int** **ListCompletedJobs**(); | Lists the ID of the completed jobs |

## StateManager

StateManager handles the transfer of state between nodes as well as provides access to local and remote state to the adaptor. This is executed on a thread which periodically exchanges state among the nodes.

|  |  |
| --- | --- |
| **int** **Initialize**(configInfo \*config,CCommProxy \*proxy,CHWMonitor \*monitor, CJobQueue \*pJobQueue); | Initializes the manger with the required components. Starts a thread with period from the config |
| **int** **UpdateRemoteState**(State &state); | Sends request to remote node for its state |
| State **GetMyState**(); | Return the local state |
| State **GetRemoteState**(); | Returns the remote state |

## TransferManager

Transfer manager takes the responsibility for handling all data transfer between the nodes including sending and receiving jobs during load balancing and also completed jobs. The APIs exposed are given below.

|  |  |
| --- | --- |
| **int** **Initialize**(CJobQueue \*pJobQueue, CCommProxy \*pProxy); | Initializes the transfer manager. No thread there. All call will occur in the adaptor thread |
| **int** **SendJobsToRemote**(**int** size); | Sends jobs to remote by fetching it from the job queue |
| **int** **RequestJobsFromRemote**(**int** size); | Requests size number of jobs form the remote node |
| **int** **AddJobsToLocalQueue** (std::vector<CJob\*> &vJobs); | Adds jobs to local queue on receiving from remote |
| **int** **RequestCompletedJobsFromRemote**(); | Requests for the completed jobs from remote |
| **int** **SendCompletedJobsToRemote**(); | Send completed jobs to remote |

## Worker

Worker is the thread thattakes each job from the queue and process. The thread is created during the initialization call whose signature is provided below. The stop thread method stops the thread.

**int** **Initialize**(CJobQueue \*pJobQueue, CHWMonitor \*pMonitor);

void StopThread();

## RPC Mechanism for communication:

We use Apache thrift for communication between the nodes as well as with the GUI. The interface files defining the communication method is provided below. All communication between the processing nodes are asynchronous. For the GUI the methods are synchronous since the server doesn’t have proxy to the GUI to send the state.

struct UIState

{

1:double cpu\_util;

2:i32 nJobsPending;

3:i32 nJobsCompleted;

4:double fThrottling;

5:double dNetwork;

}

service DynLBServer

{

oneway void SendJobsToRemote(1:i32 size, 2:list<binary> vJobs);

oneway void RequestJobsFromRemote(1:i32 nJobs);

oneway void SendStateToRemote(1:binary stateBlob);

oneway void RequestStateFromRemote();

oneway void SendCompletedJobsToRemote(1:i32 size, 2:list<binary> vJobs);

oneway void RequestCompletedJobsFromRemote();

// we will need synchronous calls during GUI impl to be called

// from java UI since server has no proxy to the UI client.

UIState GetStateInfo();

void SetThrottling(1:double throttling);

}

## Configuration

The configuration is specified in an xml file and shared among all the processes including the GUI. The xml is self-explanatory and is given below.

<DynLB >

<!-- 1024 \* 1024 \* 32 = 33554432, 1024 \* 1024 \* 16 = 16777216 -->

<Nodes>

<Node id=*"0"* type=*"local"* ip=*"172.22.156.65"* port=*"9090"* />

<Node id=*"1"* type=*"remote"* ip=*"172.22.156.5"* port=*"9090"* />

</Nodes>

<Workload size=*"33554432"* jobs=*"512"* />

<ThrottleInterface file=*"throt"* />

<HWMonitor period=*"1000"* />

<StateInfoPolicy period=*"2000"* /> <!-- ms: can also include items="network,cpu" -->

<TransferPolicy period=*"5000"* type=*"sender"* algo=*"jobCount"* /> <!-- sender,receiver,symmetric -->

<!-- algo<jobCount,jobCompletion,advanced> -->

</DynLB>

# Load balancing techniques:

## Types

### Sender Initiated:

In this scheme, if a node detects load imbalance due to it being overloaded then it sends some jobs to the remote node. To select this mode, configure the xml with type=”sender”

### Receiver Initiated:

In this scheme, if a node detects load imbalance due to the other node being overloaded then it requests for some jobs from the remote node. To select this mode, configure the xml with type=”receiver”

### Symmetric:

In this scheme, if a node detects load imbalance due to the other node being overloaded or it being overloaded then it requests or sends half of the of the jobs that will create the balance. The expectation is that the other node will also send/request half of the jobs required for load balance.

## Algorithms:

### Job count

The difference in job count is used to balance the load

### Completion Time

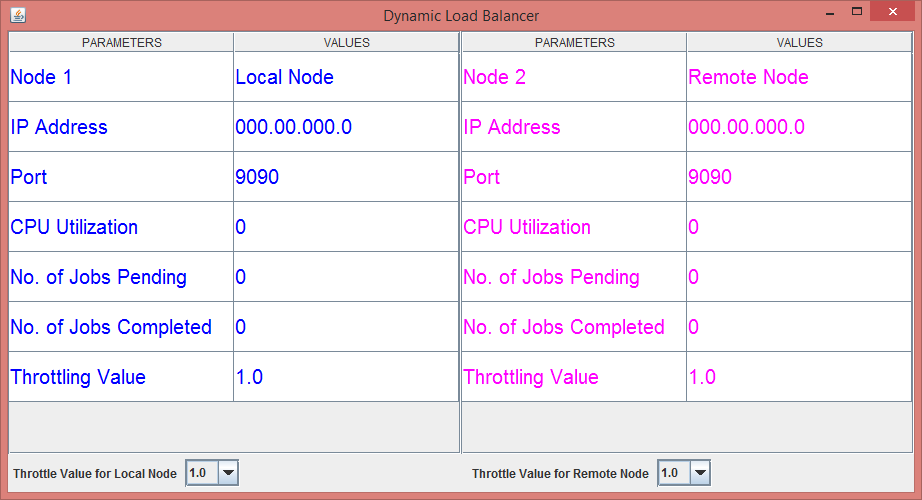
The difference in estimated completion time considering the throttling value is used to balance the load.

### Advanced

Network bandwidth is used for load balancing by figuring out if the overhead of communication is more than the job completion in the local node.

*Over head of communication* = (latency + size/bandwidth) \* (1 + packet\_loss/100)

## Graphical User Interface Implementation



This is the basic layout of the Dynamic Load Balancer. It has been modelled using Java Swing and Java AWT components like JTable, JComboBox, JFrame, etc.

It shows a comparison between the two nodes provided for the MP i.e. the local and the remote node. The comparison can be defined in terms of IP Address, Port, CPU utilization, number of jobs pending, number of jobs completed and the throttling value.

The throttling value can be increased or decreased by the user from the drop down provided for both the nodes.

Since our main coding is written in C++ and the GUI is developed in Java, thrift complier has been used to generate the required files in Java to communicate with server.

|  |  |
| --- | --- |
| **Files** | **Description** |
| DynamicGUI.java | Class containing methods to communicate to both nodes and display values on the GUI |
| Values.java | Class containing the values to be stored for each node |
| DynLBServer.java | Auto-generated file by thrift compiler |
| UIState.java | Auto-generated file by thrift compiler |

The two files DynLBServer.java and UIState.java contains methods which are used to communicate and get the values which are to be displayed on the GUI.

# Compression

We used the zlib library to implement the compression and un-compress routines.

They are declared as common function in Common.h and implemented in CJob.cpp

The APIs are

**void** **compress\_buffer**(**void** \*in\_data, size\_t in\_data\_size, std::vector<uint8\_t> &out\_data);

**void** **uncompress\_buffer**(std::vector<uint8\_t> &in\_data, std::vector<uint8\_t> &out\_data);

The pseudo code is:

1. fill next\_in with the next chunk of data you want to compress

2. set avail\_in to the number of bytes available in next\_in

3. set next\_out to where the compressed data should be written which should usually be a pointer inside your buffer that advances as you go along

4. set avail\_out to the number of bytes available in next\_out

5. call deflate

6. repeat steps 3-5 until avail\_out is non-zero (i.e. there's more room in the output buffer than zlib needs - no more data to write)

7. repeat steps 1-6 while you have data to compress

8. Eventually you call deflateEnd() and you're done.

For our use case we get a very high compression ratio (~ 900), but the cost of compressing is higher that the cost of sending the original buffer. We get higher runtimes with compression enabled.

# Empirical Results and Analysis:

## Runtime for different combination of policy type and algorithm

Time here indicates time taken to complete processing all jobs.

**Experimental parameters:**

1. throttling values 0.75(node 0) 0.25(node 1)
2. jobs = 512
3. workload = 33554432 doubles

d) load balancing period = 5 seconds

Time without any load balancing: **213.30 secs**

|  |  |  |
| --- | --- | --- |
| Type | Algorithm | Time (secs) |
| Sender Initiated | jobCount | 144 |
| Sender Initiated | jobCompletionTime | 141 |
| Receiver Initiated | jobCount | 141 |
| Receiver Initiated | jobCompletionTime | 138 |
| Symmetric | jobCount | 138 |
| Symmetric | jobCompletionTime | 141 |

Even though jobs are moved to remote node during imbalance, the time taken to serialize the jobs are send them to remote node has almost the same overhead as running the jobs locally. This has resulted in the performance with and without load balancing to be almost the same. Things would improve a lot in a multi-core environment where one of the threads can take care of load balancing while the job processing is not stalled. Also if the job in our experiment is more computationally intensive then we will see an improvement

## Effect of load balancing period on total time

Experiments are done with sender initiated type and job count algorithm

The impact of load balancing period is minimum.