MapReduce on Multi-core

Initial Project Report
COMP60990 - Research Skills and Professional Issues
2009

George Kovoor
School of Computer Science
# Table of Contents

Abstract.................................................................................................................................................. v

1 Introduction........................................................................................................................................... 1
  1.1 MapReduce and Parallelism............................................................................................................ 2
  1.2 Aims and Objectives......................................................................................................................... 3

2 Background .......................................................................................................................................... 4
  2.1 Implementation on Cluster Computing Environment ................................................................. 4
  2.2 Execution Overview........................................................................................................................ 5
  2.3 Additional Features........................................................................................................................ 7
  2.4 Implementation on Shared Memory System................................................................................... 7

3 Research Methods.............................................................................................................................. 11
  3.1 MapReduce Architecture............................................................................................................... 11
  3.2 Programming Model....................................................................................................................... 11
  3.3 Execution Overview....................................................................................................................... 13
  3.4 Design Overview............................................................................................................................ 14
  3.5 Evaluation......................................................................................................................................... 15
  3.6 Development Methodology............................................................................................................ 16
  3.7 Deliverables..................................................................................................................................... 17

4 Understanding...................................................................................................................................... 18
  4.1 Risks and Technical Challenges.................................................................................................... 18
  4.2 Optimisation Techniques and Additional Features.................................................................... 19

5 Summary............................................................................................................................................... 20

References.............................................................................................................................................. 21

Appendix A............................................................................................................................................. 23

Appendix B............................................................................................................................................. 25
List of Tables

Table 1: MapReduce core API................................................................................................. 15
Table 2: Production system specifications.................................................................................. 15
Table 3: Applications used for evaluation .................................................................................. 15
Table 4: Development tools for the project .............................................................................. 17
List of Figures

Figure 1: Correlation between CPU frequency and Moore’s Law .......................... 1
Figure 2: MapReduce execution overview .......................................................... 6
Figure 3: Phoenix execution overview ................................................................. 9
Figure 4: High-level execution overview ............................................................. 13
Abstract

MapReduce is a programming model that is used to implement the processing of a large dataset as a sequence of distributed operations. The primary advantage of this programming model is that, it allows automatic parallelisation of applications written in a functional programming style. This allows programmers with no specific knowledge of parallel programming to attain parallelism. This report primarily deals with the design, implementation and evaluation of a MapReduce framework targeting multi-core system architectures. Various optimisation techniques are considered during the design stage of the framework to make it highly efficient on shared memory system architectures.

The design and implementation of the MapReduce framework consist of two core sections, the API for MapReduce programming environment and MapReduce runtime system. The implementation of the MapReduce runtime system is specifically tailored for shared memory multi-core systems. However, the final version of the framework should provide considerable performance on any multi-core architecture.

Evaluation of the framework will be performed using standard benchmark applications from various domains. Evaluation will be based on usability, performance, scalability and additional features such as fault tolerance, monitoring and debugging capabilities provided by the framework.
1 Introduction

Moore’s law states that the numbers of transistors on a chip will double every 18 months. This has been one of the basic guiding principles in designing CPU [1]. It has been observed that the frequency of the CPU roughly follows this pattern. From late 1990 to 2004 we can observe a consistent pattern of CPU frequency doubling nearly every two years. This correlation between the CPU frequency and number of transistors no longer holds true in the current decade. One of the important trends to note is that during the period from early 1990-2004 the performance of the CPU relates directly to the frequency of the CPU. This means that doubling the CPU frequency resulted in doubling the CPU performance. Figure 1 captures this trend in the computer industry.

![Figure 1: Correlation between CPU frequency and Moore’s Law. Source taken from [2]](image)

It can be seen from Figure 1, that the CPU frequency tends to drop from early 2005. This shows that increasing the CPU frequency is limited by several factors. The main factors that limit increasing the CPU frequency beyond certain limits are the following. The size of the transistors is becoming smaller that it is approaching its physical limit. As a result designing CPU below 65nm in size has resulted in higher hardware efforts. This has a direct impact on the development time and cost. Other factors such as the power consumption and heat dissipation have direct influence on the frequency of the CPU. It has to be noted that until late 2004 increasing the performance of a CPU had direct correlation with Moore’s law. As the number of transistors doubled the frequency tend to increase proportionally, this in turn resulted in increased the power consumption and heat dissipation of the chip. The techniques such as instruction level parallelism (ILP) that is used to improve the performance have become highly complicated to be implemented effectively on a single core CPU. As a result the ILP improvements have reached its barrier referred to as ILP Wall [3].

Another major factor that accounted for restricting the CPU frequency is the limited memory bandwidth, referred as the Memory wall [4]. This has been one of the biggest challenges in developing faster CPUs in the current decade. Because the memory is clocked at much lower speed than the main CPU, increasing the CPU speed will only
result in CPU waiting for the load and store operation on the memory to complete and would result in additional memory access overhead.

Because of these barriers in developing faster sequential CPU, major CPU vendors have now turned to design the chips with multiple cores to meet the demands of high performance. The use of multi-core chip has several advantages compared to sequential chip. One of the primary advantages of using multi-core processors is that it reduces the hardware challenges involved in building high performance sequential chip. From the equation \( P = CV^2 F \), we know that decreasing the power requires decreasing the threshold voltage. On a sequential CPU, clocking at higher frequency (i.e. Increasing performance) requires minimum amount of voltage for optimal performance, this in turn results in increase in power consumption. On a multi-core chip, the performance improvement is based on processor level parallelism, whereas on a sequential chip performance improvement is based on increasing the CPU frequency [3]. Hence, a multi-core chip can deliver higher performance with the same amount of resources such as memory, power etc as that of the sequential processor chip.

1.1 MapReduce and Parallelism

The requirement for high performance in the industry has caused multi-core technology to be the mainstream in designing commercial CPU. The Microprocessor industry had turned to parallel computing to make use of efficient power consumption, higher performance, lower heat dissipation and above all reduce the challenges and risks involved in manufacturing single core processors to meet the requirement for high performance. This current trend in manufacturing multi-core for commercial microprocessors has given rise to a new set of challenges.

One of the most critical challenge is how to make full use of the entire cores on the chip. This is important to attain the benefits of multiple cores. It is forecasted by experts that the Moore’s law can be applied to the number of cores available on a chip [3]. This would imply that the programming challenges on a multi-core system would increase proportionally as the amount of parallelism required for a software doubles every year [5, 6]. This could mean that Amdahl’s law drives the performance delivered from multi-core chip.

So it is evident that in order to process more data we require more computational power, and one way to attain these computational power is to make use of distributed computing environment or a multi-core system architecture to compute the tasks in parallel. Hence, we require a programming model that can make effective use of parallelism on multi-core and distributed computing environment.

MapReduce is a programming model that provides the programmer an effective environment to attain automatic parallelism. The advantage of MapReduce model is that the programmers need not be aware about the internal implementation of attaining parallelism on a specified architecture nor do they need the knowledge of parallel programming to attain large-scale parallelism. The MapReduce runtime system handles
these internal low-level details to attain parallelism for the programmer. This ease of use feature of MapReduce framework enables the programmers to focus more on the application logic rather than dealing with low-level parallelisation details. Typically in MapReduce framework the programmer is only concerned with expressing a given problem or computation in a functional programming model. Once these functions are defined by the programmer, the runtime system takes the responsibility of automatically parallelising it on the given hardware architecture. One of the challenges of using MapReduce framework to attain parallelism is that not all the problems or computational tasks can be decomposed efficiently into the functional style model, hence it wouldn’t be a suitable model for various application domain[7]. The implementation of MapReduce framework varies significantly based on the execution environment. For example, the implementation of MapReduce on a shared memory system is different from the implementation on a distributed computing environment. However, the underlying concept of computation remains the same in both cases i.e. low level parallelism details are abstracted from the programmer.

1.2 Aims and Objectives

The aim of this project is to conduct a research in designing and developing a feasible high performance MapReduce framework for multi-core systems. The design and implementation of this framework is based on the shared memory architecture.

The framework mainly consists of the two main components, Runtime system and MapReduce API. The purpose of the runtime system is to provide automatic parallelism by handling the responsibility of dynamic thread creation, thread management, dynamic task partitioning, load balancing and scheduling. The Runtime system also provides additional features such as fault tolerance, debugging and monitoring capabilities for the framework.

The MapReduce API provides the basic interface for the programmers to decompose a given computational problem into map and reduce function. It also provides the additional features such as making use of Java Generic in defining the input and output data format so as to provide type safety and options to specify the user defined partitioning and splitting functions to attain better load balancing.

The final objective of this project is to evaluate the developed framework for its performance and usability using the standard benchmark applications selected for this project from various domains. In addition, provide a quantitative analysis of the speedup obtained, efficiency of the runtime system, scalability of the framework on different multi-core architectures, and critically evaluate the fault tolerance and debugging capabilities of the framework.
2 Background

MapReduce is a general purpose-programming paradigm developed by Google to process very large dataset in parallel on a cluster computing environment [8]. The simplicity of this programming model is that it allows the programmers to define the computational problem using functional style algorithm. The runtime system automatically parallelises this algorithm by distributing it on a large cluster of scalable nodes. The main advantage of using this model is the simplicity it provides to the programmer to attain parallelism. The programmer need not deal with any parallelisation details, instead focuses on expressing application logic or computational task using sequential Map and Reduce function. Perhaps one major challenge with this model is its flexibility in expressing complex task due to its simple and restricted interface. However study conducted by Dean and Ghemawat [8] have demonstrated numerous examples of using MapReduce model at Google in solving applications from various domains, such as indexing system, machine learning problem, data and graph extraction and reverse URL on a distributed clustered environment. Apart from automatic parallelisation, MapReduce runtime system also provides fault tolerance, status monitoring, data partitioning and scheduling functionalities that are abstracted from the programmer. This low level abstraction provides ease of use and scalable environment for attaining massive parallelism. An important aspect of this research is to identify the following objectives

- Usability of MapReduce programming model on a multi-core system.
- How the MapReduce model could be optimised to improve its performance and scalability.
- Implications of using Java threading library to attain parallelism on multi-core systems.
- Effective handling of failures on parallel execution of tasks.

There are several implementations of MapReduce framework on various system architectures. This research mainly focuses on two main implementation of MapReduce framework namely, Google’s MapReduce model for large cluster based network and shared memory model based on C/C++ using P-threads threading library [7].

2.1 Implementation on Cluster Computing Environment

The primary advantage of using MapReduce model is that it provides automatic parallelisation through functional programming style. This functional approach makes the programming model embarrassingly parallel [3, 8]. Meaning, any input to map and reduce function are computed without requiring dependency on any other elements, this eliminates the contention for locks and other synchronisation techniques that are more prone to errors. The target environment for Google’s implementation of the MapReduce framework is based on large cluster consisting of commodity PC interconnected using high-speed gigabyte switched network. It uses a specialised file system called GFS (Google File System) [9] used to manage data across the nodes. Google’s implementation
of the MapReduce programming model can be broken down into the following two main categories API and Runtime system. The API can be broadly divided into User defined and System defined functions.

2.1.1 User defined functions

User defined functions generalises the functions defined by the programmer to express the application logic. The core functions consist of Map and Reduce function. Both these functions take input argument as key/value pairs. Map function takes key/value pairs as input and produces an output containing a list of intermediate key/value pairs [8, 10]. The reduce function takes the sorted output that consists of unique intermediate key and list of values associated with it as input and performs reduction or merge operation on these set of values to produce zero or one output values per unique keys. The pseudo code given below summarises these concepts.

Map(input_key,input_value) --> list (output_key, intermediate_value)
Reduce (output_key, list (intermediate_values)) --> output_value

2.1.2 System defined functions

The system-defined function specifies the functions provided by the Runtime system to programmer. These functions mainly consist of initialisation function, used to initialise the scheduler, functions used to store the output key/value pairs into a temporary buffer and final storage location and other functions to specify optional parameters to configure scheduler.

The important feature of the MapReduce API is its restricted interface. It has been proven that use of narrow and simplified interface had helped programmers in defining the application logic more accurately, which is easier to debug and maintain [7, 8, 10]. Restricting the API had also helped in improving its usability by lowering the learning curve associated with it compared to other commercially available API for parallelism such as OpenMP [11].

2.2 Execution Overview

Programmer makes use of MapReduce API to define the application logics using User defined functions and initialise the runtime system using System defined functions. Once initialised the runtime system spawns several worker threads on the nodes, each of these worker threads are assigned either a map or a reduce task. One of the worker threads is elected to be the master/scheduler based on the configuration used by the Runtime system. Figure 2 shows the execution overview of Google’s Model.
The runtime system accepts the job submitted by the programmer. Each job consists of a set of map and reduce task. The runtime system automatically partitions the input data/file into smaller chunk size (default chunk size is 16MB to 64 MB) using splitting function. These partitioned data is assigned to map task workers distributed across the node by the master thread. Each map task worker processes the input data in parallel without any dependency. The output from the map task worker is stored locally on the system on which the task is executing. Once all the map tasks are completed, the runtime system automatically sorts the output from each map task worker so that the values from the same intermediate keys are grouped together before it is assigned to the reduce task worker. The reduce tasks are assigned to the reduce task workers by partitioning the intermediate sorted keys using the partitioning function. Partitioning function uses key hashing \((\text{hash (key)} \mod R; \text{where } R \text{ is the number of reduce task worker})\) [12] to partition the reduce task. The output from reduce task workers are stored on a GFS (Google File System) for high availability. When each reduce task completes the reduce worker renames the temporary file to the final output file specified by the user, hence multiple rename calls are executed by the reduce task worker. GFS provides atomic rename support to guarantee that the final output is consistent.

The master/scheduler is responsible for the following tasks in Google’s MapReduce model. It maintains the status information and identity for map and reduce tasks. It is responsible for transferring the location of the file from the map task worker to the reduce task worker. Delegates map or reduce task to each workers. Maintains locality by assigning tasks locally to the workers and manages worker’s termination.
2.3 Additional Features

Several features are added to optimise the performance and fault tolerance in Google’s MapReduce model. Load-balancing techniques are applied, by allowing any task to be assigned to the workers that are scattered across the cluster. This helps in achieving both dynamic load balancing and faster recovery from failed task. Efficient partitioning of the input data between 16 to 64 MB chunks helps in balancing the number of map and reduce tasks assigned to the workers. This helps in reducing the overhead due to scheduling. One of the important features implemented in Google’s model is the backup execution of remaining in-progress tasks on the available idle worker threads. Dean and Ghemawat [8] demonstrated that backup task execution had resulted in significant improvement in performance (up to 44%) for applications such as sorting. It is found that the load imbalance was the primary cause for long executing tasks.

Google’s model of MapReduce has an efficient fault tolerance mechanism in place, as the numbers of machine failures are frequent due to the large cluster size. The failures are determined by pinging the worker machines periodically. If a failure is detected on an in-progress task, the failed worker is reset to idle so that worker can be rescheduled for execution. On the other hand, a completed map tasks on a failed worker has to be re-executed, as the task becomes no longer accessible when the worker fails because output results are stored locally. The reduce task output are stored globally, so any failure of reduce task workers will not require re-executing the completed task. Another important feature of the fault tolerance mechanism is that the master monitors the input data that causes the failure, repeated failure of a particular input data causes the master to skip the re-execution of that input data, this helps in dealing with issues caused by bugs in third-party libraries.

In Google’s, model most of the optimisation options deals with reducing the overall network bandwidth consumption. This is due to high communicational cost associated between nodes in different rack in the cluster. Optimisations are applied by maintaining the locality while partitioning the data for map tasks. The master assigns map task to workers that are in close proximity to the input data and this allows data to be read locally. Use of combiner functions to perform mini reduction helps in reducing the load imbalance in reduce phase.

Monitoring options are provided as a part of debugging feature, the master maintains an internal web server that provides the status of running and non-running tasks assigned to the scheduler. Google’s model also provides support for sequential execution of task as a part of debugging process.

2.4 Implementation on Shared Memory System

This section discusses the implementation of MapReduce model know as Phoenix for a shared memory architecture developed by Ranger et al [7] from Stanford University. The underlying principle of Phoenix framework is based on Google’s MapReduce model, hence it shares several features with Google’s model. Phoenix implementation of
MapReduce is targeted for shared memory architectures on multi-core and multi-processor system rather than cluster based environment. The major difference in Phoenix implementation is that it uses threads to do the processing instead of machines/nodes. Another difference is in the mode of communication. Shared memory model uses inter thread communication whereas a distributed computing model uses network messages to communicate between nodes using RPC calls [8].

As with Google’s model, Phoenix model can be broadly classified into two sections the Runtime system and the API. Phoenix API is based on C implementation using P-threads as the threading library. The API can be sub classified into User defined and System defined functions. User defined function provides an interface to define user specific functions such as map, reduce and other functions used to configure the scheduler. System defined function includes the functions provided by the runtime system to the programmer. The fundamental structure of Phoenix API resembles Google’s MapReduce API by maintaining a narrow and simplified interface.

Phoenix Runtime system handles the low-level parallelisation details for the programmer. The Phoenix Runtime is responsible for creating and managing the threads across multiple cores. Dynamically schedules map and reduce tasks to the worker threads. Handles communication between worker threads and maintains state for each worker threads. Beside these functions the runtime system also provides additional features such as dynamic load balancing, fault tolerance, and perform various optimisations techniques such as improved locality management and dynamic scheduling.

### 2.4.1 Execution overview

User program initialises the scheduler by invoking initialisation function (`int phoenix_scheduler()`), the scheduler spawns several worker threads based on the number of cores supported by the system. Each worker thread is assigned map or reduce task by the scheduler. Figure 3 summarises the execution of Phoenix system.
The scheduler partitions the input data using the splitter function `int (splitter_t)`. Default splitter function uses cache size of the system to determine the chunk size. Partitioned data is forwarded to the map task workers assigned dynamically by the scheduler. Once the map task worker completes processing the input data, it calls the emit intermediate () method to store the output to a buffer (intermediate map queue). The partition function acts as a combiner function that combines the values from the same intermediate key into a set of key/value pairs with unique key for each set of values. Reduce task starts only when the entire map tasks completes, this is found to be the case with Google’s MapReduce model.

In the reduce phase the reduce tasks are assigned dynamically by the scheduler based on the partitioning function. Default partitioning function uses key hashing (hash (key) mod R), this is similar to the partition function used in Google’s MapReduce model, Phoenix system also provides an option for the programmer to specify the partitioning function to be used. The reduce task worker computes the data from the intermediate map queue, the output from the reduce task worker is stored in the reduce queue. Each reduce task worker maintains separate reduce queue, this is done to minimise the contention for the queue.

Merge phase is considered optional, in this phase the output from the reduce queue is merged together to produce a single output. In Google’s implementation, this phase is achieved by making recursive call to MapReduce. The overhead associated with merge phase is proportional to the size of each reduce queues. Merge takes place in $\log_2 (N/2)$ steps where N is the number of worker threads performing merge operation. Study conducted by Ranger et al [7] have reported that merge phase is not required for
most of the application and the overhead associated with it is comparatively lower than making recursive call to MapReduce function. Hence, we can argue that the use of merge stage can help minimise additional communication and scheduling overheads.

2.4.2 Additional features

Phoenix runtime system provides many additional features such as load balancing, locality optimisation and fault tolerance. The following techniques are employed by the Runtime system to optimise overall performance of the framework.

The use of dynamic scheduling of map and reduce tasks has resulted in well-balanced load among the worker threads. The default splitting function that is used to partition the input data uses the L1 data cache size to determine the optimal chunk size of the partition. This has resulted in optimising the locality. Phoenix further improves the locality by using pre-fetching engine to fetch in advance the next input data to be processed into the L2 cache of the system. Hardware compression mechanism used in Phoenix helps to reduce the size of the intermediate output data and memory bandwidth utilisation. Phoenix also provides an option for the programmers to specify a custom partitioning function. The default partitioning function uses key hashing for distribution of task. It has been found that since the keys can be associated with different values the computation required by various keys may differ resulting in significant load imbalance. The option to specify custom partition function would help to mitigate the overhead due to load imbalance as the programmer can provide partitioning function based on the application specific knowledge [7].

Phoenix runtime detects faults through timeout mechanism. It uses execution time of similar task as a measure to determine the maximum duration required to execute a similar task. If a worker thread is found to be faulty (i.e. exceeds the maximum duration allocated for the particular task), the runtime system re-executes the affected task with a separate output buffer. A new output buffer is used for each worker threads as this helps in minimising contention for data. The scheduler decides if the fault in the worker thread is transient or permanent fault, based on the frequency of failed attempts. Initially any failure by the worker thread will default to transient error. If the overall failure on a worker threads increases beyond the threshold limit then that worker thread is assumed to have a permanent failure. Worker threads registered with permanent fault will have no further task assigned to it.

Limitations of the fault tolerance mechanism implemented in Phoenix system are identified as follows, lack of support to determine if a task is completed successfully by a worker thread, single point of failure for the scheduler, in other words a fault with the scheduler will cause the application to terminate prematurely.
3 Research Methods

This research involves developing a MapReduce framework that can be used to simplify parallel programming for processing large data set on shared memory multi-core architectures. This framework will include the necessary optimisation features to make it highly efficient and scalable. One of the important aspects of this research is to identify and evaluate the suitability, ease of use and efficiency of this programming model on multi-core systems.

3.1 MapReduce Architecture

The high level structure of this framework resembles Phoenix implementation [7] of MapReduce model, since it deals with shared memory multi-core architecture. This model has adapted several optimisation features from various implementations of MapReduce model such as Google’s MapReduce model on large cluster network [8], Implementation on Cell processors by University of Wisconsin-Madison [13] and Microsoft’s DryadLINQ an implementation on cluster environment [14].

The high level architecture of the proposed model can be subdivided into two parts the Runtime system and the API. The Runtime system is based on master/slave architecture. This model uses a single scheduler/master thread responsible for the following, Creating, managing and terminating the slave/worker threads, Schedules the map and reduce task to the worker threads, Maintains the state and buffer management for each worker threads and Handles the communication between these worker threads.

In this model, scheduler is a single point of failure, there is no failure recovery mechanism currently employed for the scheduler. The programmer interacts with the framework through the interface provided by the scheduler. The programmer uses the API to express the application logic in a sequential implementation of map and reduce tasks. On submitting the MapReduce job to the scheduler, the tasks are automatically distributed and processed in parallel by the runtime system.

3.2 Programming Model

Programmer defines the application logic using Map and Reduce function. The Map function takes key/value pair as input and emits intermediate key/value pair as output. The input to the Map function is processed independently without any dependency on other subset of data. This allows the map functions to be executed in parallel by different worker threads. The output from the map tasks are written to map queues using emit_intermediate function, this is a temporary storage area for map tasks.

Once the execution of Map phase is completed, the runtime system sorts the intermediate key output from each map queues so that values from the same keys are grouped together to form a unique key/value pair, this avoid duplicate intermediate key/value pairs. The framework partitions the sorted unique key/value pairs among the
available reduce task workers. In this model both sorting and partitioning takes place at
the barrier, partitioning is performed using key hashing [12].

The reduce phase starts once all the map tasks are completed. Input to the reduce
task is the unique sorted intermediate key/value pairs. As with the map task, the reduce
tasks are executed in parallel without requiring any synchronisation mechanism. The
reduce task usually performs some kind of reduction such as summation, sorting, and
merging operations. The output from the reduce tasks can result in zero or one key/value
pair, the final output from reduce task worker is written to reduce queue using emit
function. In some cases if the output is more than one key/value pair then a merge is
performed on the value of the reduce queue to produce a single output. Pseudo code
example of using the MapReduce model for counting the occurrence of a word in a
document is given below. In the example below reduction is performed by summing the
values of the unique key.

Document name: “namelist”
Document content: “George Thomas Kovoor George Kovoor Thomas Kovoor George”
Expected output: George 3, Thomas 2, Kovoor 3

Map Function:
/*key is the document name Namelist
value is the content of Namelist*/
Map(String key, String value)
{For each word w in value
emit_intermediate(w,1);
}

Map(namelist, “George Thomas Kovoor George Kovoor Thomas Kovoor George”)

Map task Output (intermediate key/value pairs):
[George,1],[Thomas,1][Kovoor,1][George,1],[Kovoor,1],[Thomas,1],[George,1],[Kovoor,1],

Note: output order is not guaranteed.

Outputs from the map task are sorted by the Runtime system, so that
values of the same keys are grouped together.

Sorted Output:
[George,1,1,1],[Thomas,1,1],[Kovoor,1,1,1]

Reduce Functions:
/*key is the unique sorted intermediate key
value is the value associated with the key*/
Reduce (String key, List values){
int result=0
For each v in values
result+=v;
emit(key,result)}

Reduce(“George”,[1,1,1])
Reduce task output:
George 3;
3.3 Execution Overview

The scheduler is responsible for controlling various activities of the runtime system. This includes tasks such as creating and managing worker threads, delegating map and reduce task to the worker threads, maintaining the status of worker threads, handle buffer management, and also communication and termination of each worker thread. Application (User program) initialises the scheduler using the initialise method, this starts the MapReduce runtime system. The scheduler forks multiple worker threads based on the arguments passed to the runtime system using the User defined API. By default, spawning of the worker threads is based on the number of cores supported by the system. Figure 4 illustrates the high-level execution overview of the proposed framework.

![High-level execution overview diagram](image)

**Figure 4: High-level execution overview**

In the Map phase, each worker is dynamically assigned a map task by the scheduler. For each map task worker, the scheduler invokes the splitter function to
partition the input data into equally sized smaller chunks of data. The partitioned data is passed as an argument to the map task workers. Each map task worker outputs a set of intermediate key/value pair. The output from each worker is stored in its own temporary buffer. The scheduler deals with managing the buffer allocated to each worker. Once the Map phase is completed, the Runtime system starts sorting the intermediate key/value pair output from each map task worker. The scheduler determines the completion of the map phase when all the map task workers arrives at the barrier.

Sorting is performed based on the sorting function used by the framework, it ensures that the values from the same keys are grouped together so that the output of sorting will result in unique key per set of values, this step ensures that all duplicate keys will be merged together. The sorted key is then partitioned using the partition function. The partition function partitions the sorted intermediate key/value pair into small chunks such that each partition contains unique key and it’s associated values.

Reduce phase begins once the partitioning of the intermediate key/value pairs is completed. Hence there is some delay in starting reduce phase as the sorting and partitioning function needs to be completed. The scheduler dynamically assigns reduce tasks to the worker. Each reduce task worker uses the partitioned data as input, since each reduce task worker is assigned data related to a single unique key, there will be some load imbalance during reduce phase. The output from the reduce task workers are stored in the reduce queue.

The merge phase begins when all the reduce task worker have reached the barrier, this is an optional stage and is required based on the application logic. During merge phase, the output from each reduce queue is merged in $\log_2 (P/2)$ steps [7, 15] (where $P$ is the number of worker threads used during parallel merge operation), this is similar to the merging performed in Phoenix implementation [7].

### 3.4 Design Overview

The fundamental design of the framework is based on Phoenix MapReduce model [7], hence it has several structural resemblance to Phoenix API. As with phoenix model this model can be broadly sub divided into two main sections the Runtime system and the API. The Runtime system deals with automatic parallelisation of the application logic expressed using Map and Reduce function. The API provides the programmers the interface to communicate with the Runtime system. The API consists of System defined Interfaces and User defined Interfaces. System defined interfaces contains concrete classes provided by the API to the programmer. These interfaces are mostly used to manage the Runtime system. The User defined interface provides interfaces that programmers can use to define an application logic and for specifying optional arguments to the scheduler. An overview of the core API is summarised in the Table 1.
MapReduce core API

<table>
<thead>
<tr>
<th>System Defined Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class MapReduceFactory</strong></td>
</tr>
<tr>
<td>Singleton class that provides an instance of the Scheduler. Responsible for the initialisation of runtime system.</td>
</tr>
<tr>
<td><strong>Class Scheduler</strong></td>
</tr>
<tr>
<td>Represents the scheduler / master in MapReduce model. Provides an instance of JobConfig, and is responsible for managing MapReduce task, such as job submission and termination.</td>
</tr>
<tr>
<td><strong>Class JobConfig</strong></td>
</tr>
<tr>
<td>Represents the configuration options for MapReduce tasks. Instance of this class is passed as an argument to the scheduler. Provides methods to store the output of map and reduce tasks to a buffer.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User Defined Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interface Mapper</strong></td>
</tr>
<tr>
<td>Defines Map tasks.</td>
</tr>
<tr>
<td><strong>Interface Reducer</strong></td>
</tr>
<tr>
<td>Defines Reduce tasks.</td>
</tr>
<tr>
<td><strong>Interface Splitter</strong></td>
</tr>
<tr>
<td>Defines partitioning functions used to split the input data.</td>
</tr>
<tr>
<td><strong>Interface Combiner</strong></td>
</tr>
<tr>
<td>Defines grouping function for Map tasks.</td>
</tr>
</tbody>
</table>

Table 1: MapReduce core API

3.5 Evaluation

Evaluation will be performed on shared memory Multi-core architectures. The purpose of this evaluation is to analyse the usability, performance and efficiency of the framework on various Multi-core systems. The following systems are considered for the final evaluation, Niagara T2 (UltraSPARC T2) and Nehalem (Core i7). The benchmark applications selected for evaluation are executed on the production system for quantitative analysis of overhead. Table 3 lists the applications and Table 2 summarises the production system specifications used for evaluation.

<table>
<thead>
<tr>
<th>Code Name</th>
<th>Niagara T2</th>
<th>Nehalem</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU type</td>
<td>UltraSPARC T2</td>
<td>Core i7</td>
</tr>
<tr>
<td>Cores</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Threads/Core</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Threads/CPU</td>
<td>64</td>
<td>8</td>
</tr>
<tr>
<td>Cache</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1</td>
<td>16 KB I-Cache</td>
<td>32 KB</td>
</tr>
<tr>
<td>L2</td>
<td>8 KB D-Cache</td>
<td></td>
</tr>
<tr>
<td>L3</td>
<td></td>
<td>8 MB</td>
</tr>
<tr>
<td>CPU frequency</td>
<td>1.4 GHz</td>
<td>2.6 GHz</td>
</tr>
</tbody>
</table>

Table 2: Production system specifications

<table>
<thead>
<tr>
<th>Applications</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>WordCount</td>
<td>Counts the occurrences of a word in the document.</td>
<td>Enterprise</td>
</tr>
<tr>
<td>Distributed Grep</td>
<td>Lists the number of matches for a given pattern in the documents.</td>
<td>Enterprise</td>
</tr>
<tr>
<td>Reverse Index</td>
<td>Computes the number of links pointing to a file.</td>
<td>Enterprise</td>
</tr>
<tr>
<td>Sort</td>
<td>Sorts a given data.</td>
<td>Enterprise</td>
</tr>
<tr>
<td>Dense Matrix Multiplication</td>
<td>Performs dense matrix multiplication.</td>
<td>Scientific</td>
</tr>
</tbody>
</table>

Table 3: Applications used for evaluation
Evaluation will be performed using applications from various domains, mainly enterprise and scientific domains. Applications considered for evaluation is comprised of the following (Word Count, Distributed Grep, Sort, Reverse index and Matrix multiplication). Other applications provided with Hadoop MapReduce model [16] are also considered for evaluation. Applications are evaluated based on sequential version and parallel version of the code. A quantitative analysis of overhead is provided as a part of evaluation, which includes plotting temporal performance, efficiency and speedup for the parallel version of the code. System timers are used to measure the execution time for the subroutines. Evaluation of the API is based on its usability and functionalities to express a given application logic.

3.6 Development Methodology

Development methodology for this project will be based on a combination of XP (extreme Programming) and Agile Model Driven Development (AMDD). Use of XP helps in taking advantage of test driven development and refactoring during software development life cycle and will aid in developing well-designed and neater code with fewer bugs [17]. On the other hand, use of agile model driven development helps in effective modelling of the various components of the framework. Each component in the framework is developed iteratively, each iteration involves designing, coding, testing and integration [18].

Detailed project plan is provide in Appendix A, which will be used to measure and control the overall progress of the research project. Proposed implementation phase for the MapReduce framework extends from 1/06/09 to 20/7/09, due to the short timeframe available priority will be given for implementing the core components in the framework these include the intrinsic features of the scheduler and the core API mentioned in Table 1. Each of these components will be implemented in several small iterations; full integration of all the components is scheduled to take place within the implementation phase. Implementation and initial testing for this project will be carried out on a heterogeneous networked environment with single core and dual core machines. The implementation phase is followed by the evaluation phase. This includes executing the pilot project on the production system (Table 2) and analysing the result.

Once the framework is finalised and evaluation phase is completed, the dissertation write up phase beings from 27/7/09 to 7/9/09 a period of seven weeks is allocated for final dissertation write up, the timeframe allocated for dissertation write up is quite flexible. Due to the complex nature of this project and the short timeframe available some tasks may need to be revised if found to be highly complicated, in such situation the project plan will be altered to adapted the new changes during development.

Project development will be carried out using Java 6; this makes the framework portable to different hardware architecture. Other major advantages of using Java 6 includes, support for high performance concurrency package, use of Generic for improved type safety and enhanced monitoring and management capabilities. Table 4 lists the development tools considered for the project.
<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Languages</td>
<td>Java 6, Jython 2.4</td>
</tr>
<tr>
<td>IDE</td>
<td>Eclipse (Ganymede)</td>
</tr>
<tr>
<td>Testing</td>
<td>JUnit 4&lt;br&gt;JMock 2&lt;br&gt;Eclipse TPTP (testing concurrency related issues)</td>
</tr>
<tr>
<td>Debugger</td>
<td>Eclipse TPTP (Test and Performance Tool Platform) plug-in&lt;br&gt;JDK 1.6 toolkit&lt;br&gt;JProfiler 5</td>
</tr>
<tr>
<td>Build System</td>
<td>Ant Build system</td>
</tr>
<tr>
<td>Dependency Management</td>
<td>Maven</td>
</tr>
<tr>
<td>Code Analyser</td>
<td>PMD Static code analyser&lt;br&gt;CAP Dependency analyser</td>
</tr>
</tbody>
</table>

Table 4: Development tools for the project

3.7 Deliverables

Deliverables for the project includes a dissertation written as per university guidelines along with complete source code for the implemented framework. The dissertation will provide a detailed investigation on the design and implementation of the MapReduce framework with quantitative analysis of the result. The source code for the framework consist of, source for the Runtime system, MapReduce API, Java doc for the API, and source for the applications that are used for evaluation (example codes).
4 Understanding

In order to make effective use of the multi-core systems software engineers must make use of parallelism. Parallel programming is inherently complex and hard to debug due to in-deterministic behaviour, hence it is prone to various kind of concurrency errors [19]. This has caused trends in parallel programming community to focus on a programming model that handles the low-level parallelisation details for the programmer.

These trends in parallel programming include automatic parallelisation, transactional memory and different implementation of MapReduce based programming models such as Yahoo’s Pig project [20], and Google’s Sawzall [21]. MapReduce framework proposed for this project is a programming model that helps in attaining auto parallelisation on multi-core systems, by hiding low-level parallelisation details from the programmer. It is this abstraction of low-level parallelisation details that helps in maximising the programming efficiency of the programmers by enabling them to focus more on the application logic rather than on the parallelisation details.

4.1 Risks and Technical Challenges

MapReduce programming model has several advantages such as auto parallelisation, ease of use, scalability, fault tolerance, and dynamic load balancing, are some of the common features. Although this programming model helps in reducing the complexity associated with attaining parallelism on multi-core and distributed environment, it has to be noted that it is not a complete generic programming model [7, 13]. There are certain tradeoffs that the programmers should be aware while using this model.

To attain parallelism, the programmers need to decompose the application logic into Map and Reduce function and provide a sequential implementation of these functions. In some cases not all the computational problems can be broken down into Map and Reduce function, hence such applications would not benefit from using the MapReduce model. Other major shortcoming of this model is that it lacks the ability to validate data by defining a logical relationship using schema [22].

Speedup obtained by using MapReduce programming model will be dominated by synchronisation and scheduling overhead of the worker threads. Other overheads such as insufficient parallelism and load imbalance are likely to occur based on the implementation of the application’s logic. Since the scheduler executes the splitting and partitioning functions, there will be some overhead due to insufficient parallelism, as both the functions are executed sequentially. As it can be seen from Figure 4, the Reduce phase will result in load imbalance because the computations performed by each reduce task worker varies significantly based on the number of values associated with each unique key.

The barrier stage in the model shown in Figure 4 will add some amount of synchronisation overhead, this is due to the fact that reduce phase cannot start until the
map phase is completed. The most significant overhead that is likely to occur will be due to scheduling of tasks (Map/Reduce tasks) to each worker threads. The primary risks involved in scheduling the tasks includes deciding on the appropriate partitioning function [10]. In the MapReduce model, we are taking advantage of fine-grained decomposition of the input data using the splitting function[23]. The partitioning function used in this model is based on key hashing [12]. Although the use of key hashing has been proven to provide good load balance in Google’s cluster computing environment [8], Ranger et al have proven that in certain cases it is found to be inefficient, if the computation differs significantly based on the input keys [7].

Other challenges involve identifying the optimal number of worker threads required to perform a set of tasks. The risk involved in forking inappropriate numbers of threads is that it will result in load imbalance and additional communicational costs that will impact the overall speedup [24]. Identifying the appropriate partitioning scheme for input data and deciding on the chunk size of the data unit is a significant factor in attaining good performance. The use of inappropriate chunk size will have considerable impact on the memory bandwidth and will result in poor cache locality. Setting the chunk size too large will result in overhead due to cache misses and setting the chunk size too small can result in increase contention for memory bandwidth [24]. Buffer management used in this model will have additional risk due to contention for data.

4.2 Optimisation Techniques and Additional Features

Several optimisation features have been considered for this project by determining efficient solutions to the identified risks, which will improve the overall speedup of the framework. This section provides the rationale for choosing certain techniques and enhancement features for the framework. It can be argued that Java program has longer instruction path than other statically compiled programming languages, and would result it slower performance. But this need not be the case as MapReduce model is considered to be extremely parallel and would result in considerable speedup even under suboptimal situations. Moreover, it is not clear at this moment what benefit the run time optimisation feature of HotSpot JIT compiler will provide to this model. These factors will be analysed during the evaluation phase of the project.

To make this programming model more flexible in addressing the issue with recursive calls to MapReduce operation, multiple calls to the MapReduce operation could be submitted to the scheduler in one invocation. This resembles the job chaining feature used in Hadoop (Cluster based implementation of MapReduce)[16]. The benefit is that all the worker threads and buffers created during the first MapReduce operation are reused, and hence would reduce the overhead associated with spawning of additional worker threads and re-initialisation of the buffer.

Providing an optional interface to the programmers to specify application specific splitting and partitioning functions can solve issues with default partitioning and splitting
The runtime system could make use of these user specified functions to achieve more efficient load balancing based on the distribution of values for specific application.

The risk due to poor cache locality and memory bandwidth contention can be avoided by providing system specific details such as the size of L1 and L2 cache size. The runtime system could then make use of the cache size information to decide on the optimal chunk size. The use of Java Caching System (JCS) could also be considered during development to improve the performance by prefetching. When using this technique the contention for memory bandwidth is also taken into account.

Monitoring the status of worker threads and dynamically allocating task based on the current usage could improve load balancing. Additional features will include, some level of fault tolerance support for the worker threads, this feature would help to handle situation when a task takes too long to complete or if the worker threads stops responding to the scheduler. Usability feature of the framework can be improved by providing default implementations for reduce task such as summation, sorting and merging so that the programmer need to specify only a sequential implementation of map task.

Debugging features would include, the ability to execute a given MapReduce task sequentially, this would help the programmers to verify the result of parallel execution. Other functionality would include logging the status of running and non-running tasks. Appendix B illustrates the format considered for monitoring the running and non-running tasks.

5 Summary

This project deals with developing MapReduce framework specifically for multi-core systems, taking advantage of current trends in parallelism. The efficiency of the model will depend on the granularity of the tasks executed by the worker threads and the usability of the API by maintaining a simple and restricted interface. Even if the framework were able to attain a speedup of 30%, it would still be beneficial to the programmer as it provides simple and efficient auto-parallelism. It is evident that the advantages provided by the framework, would clearly outweigh the disadvantages associated with it.
References

Appendix A

Gantt Chart

Project plan is divided into two sections, university weekdays and summer vacation period. This strategy will allow me to more efficiently organise my work so that I can allocate more hours to tasks that are considered time-consuming during the vacation period. The major design stage of the framework takes place during the university weekdays and development and evaluation phase progresses to the vacation period.

### Part –A (Summer Vacation)

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Duration in Weeks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7/9/09 31/8/09 24/8/09 17/8/09 10/8/09 3/8/09 27/7/09 20/7/09 13/7/09</td>
</tr>
<tr>
<td><strong>Dissertation Write-up</strong></td>
<td></td>
</tr>
<tr>
<td>Submission preparations</td>
<td></td>
</tr>
<tr>
<td>Finalise chapters</td>
<td></td>
</tr>
<tr>
<td>Review report chapters</td>
<td></td>
</tr>
<tr>
<td>Draft report</td>
<td></td>
</tr>
<tr>
<td><strong>Evaluation</strong></td>
<td></td>
</tr>
<tr>
<td>Gather feedback on API</td>
<td></td>
</tr>
<tr>
<td>Performance evaluation</td>
<td></td>
</tr>
<tr>
<td>Analysis of test result /overhead</td>
<td></td>
</tr>
<tr>
<td>Pilot project on production systems</td>
<td></td>
</tr>
<tr>
<td><strong>Implementation Phase</strong></td>
<td></td>
</tr>
<tr>
<td>Document the API</td>
<td></td>
</tr>
<tr>
<td>Finalise system</td>
<td></td>
</tr>
<tr>
<td>Integration testing</td>
<td></td>
</tr>
<tr>
<td>Module /API development</td>
<td></td>
</tr>
<tr>
<td>Module /API design</td>
<td></td>
</tr>
<tr>
<td><strong>Technical Preparation</strong></td>
<td></td>
</tr>
<tr>
<td>Identify core modules / API</td>
<td></td>
</tr>
<tr>
<td>Investigate on optimisation techniques</td>
<td></td>
</tr>
<tr>
<td>Functional requirement analysis</td>
<td></td>
</tr>
</tbody>
</table>
# Part B (University Week Days)

## Durations in Weeks

<table>
<thead>
<tr>
<th>Tasks</th>
<th>University Week Days</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dissertation Write-up</strong></td>
<td></td>
</tr>
<tr>
<td>Submission preparations</td>
<td></td>
</tr>
<tr>
<td>Finalise chapters</td>
<td></td>
</tr>
<tr>
<td>Review report chapters</td>
<td></td>
</tr>
<tr>
<td>Draft report</td>
<td></td>
</tr>
<tr>
<td><strong>Evaluation</strong></td>
<td></td>
</tr>
<tr>
<td>Gather feedback on API</td>
<td></td>
</tr>
<tr>
<td>Performance evaluation</td>
<td></td>
</tr>
<tr>
<td>Analysis of test result /overhead</td>
<td></td>
</tr>
<tr>
<td>Pilot project on production sys</td>
<td></td>
</tr>
<tr>
<td><strong>Implementation Phase</strong></td>
<td></td>
</tr>
<tr>
<td>Document the API</td>
<td></td>
</tr>
<tr>
<td>Finalise system</td>
<td></td>
</tr>
<tr>
<td>Integration testing</td>
<td></td>
</tr>
<tr>
<td>Module /API development</td>
<td></td>
</tr>
<tr>
<td>Module /API design</td>
<td></td>
</tr>
<tr>
<td><strong>Technical Preparation</strong></td>
<td></td>
</tr>
<tr>
<td>Identify core modules / API</td>
<td></td>
</tr>
<tr>
<td>Investigate optimisation technique</td>
<td></td>
</tr>
<tr>
<td>Functional requirement analysis</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B

Proposed format for displaying the status for running and non-running tasks.

**Non-Running Tasks**

<table>
<thead>
<tr>
<th>Non Running Tasks</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task1</td>
<td>Completed</td>
</tr>
<tr>
<td>Task2</td>
<td>Failed</td>
</tr>
<tr>
<td>Task3</td>
<td>Pending</td>
</tr>
</tbody>
</table>

**Running Tasks**

<table>
<thead>
<tr>
<th>Running Tasks</th>
<th>Duration (sec)</th>
<th>Status</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskName_ThreadName_m</td>
<td>120</td>
<td>Success</td>
<td></td>
</tr>
<tr>
<td>Task1_Thread2_m</td>
<td>34</td>
<td>In-progress</td>
<td></td>
</tr>
<tr>
<td>Task2_Thread1_r</td>
<td>23</td>
<td>Failed</td>
<td>Timeout</td>
</tr>
</tbody>
</table>

Running task table displays the status of the running tasks. Each task is represented by task name and the identification of the worker thread executing the task. The letter ‘m’ and ‘r’ are suffixed to the task name to denote if the task is a map task or a reduce task. Following state can be assigned to a running task, (Success, In-progress, Failed, and Pending). The number of attempts on a given task can also be displayed under the Running task table for debugging purposes. The non-running task table displays the status of inactive tasks.