A Flexible Distributed Runtime System

Hammurabi Mendes, Dimitar Todorov

December 15, 2010

1 Introduction

There is evidence that high-performance computing power in the forthcoming years is going to be provided by (1) an increasing number of computing units in computer processors; and (2) large clusters of commodity machines arranged in single-managed administrative domains.

Although the computer industry is still subject to Moore's law, which states that transistor availability doubles every about 18 months, they are now being provided in the form of an increasing number of computing units, presented either in a single computer processor or across different processors in a non-uniform memory access setting. This is due to physical limitations in constructing machines with faster clocks (heat issues, communication delays, etc), and also to the complexity related to providing increased instruction-level parallelism. Besides CPU processors, this trend is highly visible in modern graphical processing units (GPUs).

Moreover, with increasing adoption of the "software-as-service" paradigm, allied to the gradual price drops and capacity increase of commodity machines, cluster computing has been put into the spotlight. It is known that computer clusters provide high computational capacity. In fact, most high performance computing machines are actually clusters of commodity machines employing special runtime frameworks to leverage computing power.

An important consequence of this shift of paradigm is that the single-threaded computational illusion that has been provided to programmers for years is now broken. Programmers must be aware of which parts of their programs are or should be executed in parallel if they want to leverage maximum performance from the underlying computer system. Most importantly, they must be aware of the structure/architecture of the underlying system, be it a single shared-memory machine or a cluster of autonomous machines.

At present, only by being aware of the underlying system, applications might perform either an effective shared memory synchronization or the appropriate task scheduling across different machines. In such hybrid computational environments, a crucial decision is how tasks should be placed, and also how they should communicate. Ignoring such issues might severely hurt performance by unnecessarily abusing resources such as persistent storage or network bandwidth. The application developer, or the runtime system scheduler responsible for ultimately running tasks in the distributed system, needs to carefully setup how tasks should be placed and take advantage of more efficient communication mechanisms (such as shared memory).

This report presents a system called `Hammr`, which allows the specification of distributed applications in a simple manner, still providing the flexibility of specifying general directed acyclic graph application workflows for a distributed application. The motivation for this work is to build a foundation where research ideas and prototypes could be implemented over easily and cleanly. Among these ideas, are the support for heterogeneous cluster environments and GPU processor awareness, which, as discussed above, seem promising future trends for high performance computing. Besides describing the system architecture and properties, we also highlight its design structure and its client interface in this text.

2 Related Work

One of the earliest systems for distributed execution of applications is Condor [4]. Like our system, Condor aims high-throughput computing obtained from clusters of workstations, but it is also designed to leverage computing power from idle client workstations. Condor has an expressive *matchmaking* system, that couples system offers with resource requests from clients.

Paradigms such as Google's MapReduce [2] have gained attention by providing high computational power by means of a relatively uncomplicated infrastructure for running parallel applications. The programmer specifies the tasks that comprise an application and the system takes care of issues related to fault tolerance, scheduling, and other problems that arise when applications are run in a distributed setting. However, the MapReduce paradigm implies in a fixed communication structure, which might be inadequate for some applications.

Improvements to the MapReduce paradigm, like MapReduce Online [1], allow greater flexibility related to the means by which distributed applications can communicate, but the communication structure is still fixed.

Dryad [3] proposes a more general framework for executing such applications. Instead of predefining the communication graph, it is user-defined (or automatically generated by a compiler), and therefore a greater number of tasks could be expressed on top of such platform. However, as Dryad is not freely available, we opted to implement a distributed runtime system, establishing a foundation where improvements and research prototypes can be built upon.

3 Architecture

In this section we describe the overall system design, touching implementation details whenever relevant. The system was developed in Java, using the Java RMI technology, in about 5000 lines of code. We also provide a module that allows the execution of C/C++ applications in a distributed setting, implemented itself in C in about 800 lines of code.

The system is composed by a single *manager* and many *launchers*. The manager maintains a list of active launchers, and it is responsible for interpreting *application specifications* given by clients and for scheduling the corresponding applications across the launchers. The launchers receive tasks from the manager and run these tasks locally. In other words, they ultimately execute pieces of the application specified by the clients. The general architecture is shown in Fig. 1.

The clients locate the manager through a well-known distributed name server, here called *registry*. The launchers also locate the manager, and inform their presence to it, through the registry. All the communication between the clients, the manager, and the launchers, is made through Java RMI.

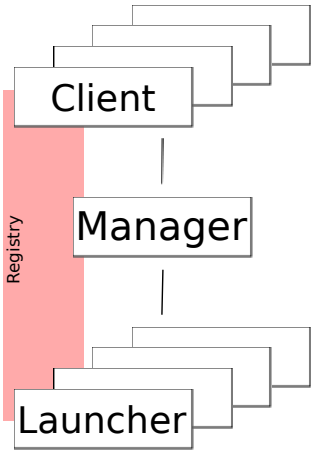


Figure 1: Multiple clients submit application specifications to the manager, that maintains a list of active launchers, which execute pieces of the submitted applications. Through a well-known registry, clients and launchers locate the manager.

3.1 Application Specification

The application specification is a graph object in which vertexes, called *nodes* here, and edges, called *connections* or *channels* here, are specified and serialized by the client and submitted to the manager. The graph needs to be a directed acyclic graph (DAG), otherwise it is rejected by the manager.

In Fig. 2, we provide an example of how clients specify an application and submit them to the manager. In the following subsections, this code will be used to explain the client-visible system features.

```
Manager manager = RMIHelper.locateRemoteObject(registryLocation, "Manager");

ApplicationSpecification applicationSpecification =
        new ApplicationSpecification("appname", "/path/superdir");

Node[] nodes = new Node[7];

for(int i = 0; i <= 1; i++) nodes[i] = new Filter();
for(int i = 2; i <= 3; i++) nodes[i] = new Copier();
for(int i = 4; i <= 5; i++) nodes[i] = new Mixer();
nodes[6] = new Relayer();

applicationSpecification.insertNodes(nodes);

for(int i = 0; i <= 1; i++)
        applicationSpecification.addInitial(nodes[i], "input-" + i + ".dat");

applicationSpecification.addFinal(nodes[6], "output.dat");

applicationSpecification.insertEdges(nodes[0...1], nodes[2...3], FILE, 1);
applicationSpecification.insertEdges(nodes[2...3], nodes[4...5], SHM);
applicationSpecification.insertEdges(nodes[4...5], nodes[6], TCP);

manager.registerApplication(applicationSpecification);
```

Figure 2: How clients submit application specifications to the manager.

3.2 The Nodes

The client-specified nodes are objects that inherit from the system-provided `Node` class, which provides useful routines to the client applications, specially in regard to communication. The system-provided `Node` class, and therefore the client-specified nodes, are actual programs that implement the Java `Runnable` interface.

When the client inserts a node in the application specification, such node gains a default name. The client could also specify other meaningful names to special nodes if desired.

In Tab. 1, we discuss the most important `Node` functions that inherited nodes benefit from. In Fig. 3, we show how client nodes are designed. These are created and inserted into the application specification by the client – see Fig. 2. We invite the reader to notice the clean design and simplicity of the application specification process, and of the client-specified node code.

3.3 The Connections

The connections are specified through the following routine:

`insertEdges(group1, group2, type, [multiplicity])`.

Method name	Method description
`getInputChannelNames()`	Return the names of all incoming neighbors.
`getOutpuChannelNames()`	Return the names of all outgoing neighbors.
`read(name)`	Read a record from input incoming neighbor called "name", blocking if necessary.
`readSomeone()`	Read a record from any incoming neighbor with records available, blocking if necessary.
`write(record, name)`	Write a record to outgoing neighbor called "name".
`writeSomeone(record)`	Write a record to some random outgoing neighbor.
`writeEveryone(record)`	Write a record to all the outgoing neighbors.

Table 1: Important methods provided by class `Node`.

```
public class Relayer extends Node {
        public void run() {
                ChannelElement channelElement;

                while(true) {
                        channelElement = readSomeone();

                        if(channelElement == null) {
                                break;
                        }

                        writeSomeone(channelElement);
                }

                closeOutputs();
        }
}
```

Figure 3: Simple information relayer implemented on top of the system.

The first two arguments are two groups of nodes. The third argument, which can be one of {SHM, TCP, FILE}, specifies whether the communication established between members of `group1` and `group2` is, respectively, a shared-memory pipe, a TCP connection or a runtime-produced file:

Shared memory. This indicates that associated nodes should run in parallel on the same machine, and that they should communicate through a shared-memory pipe;

TCP connections. This indicates that associated nodes can run in different machines, but they should be scheduled concurrently time in order to establish a stream connection;

Filesystem locations. This indicates that associated nodes can run in different machines, and that they could execute in different times.

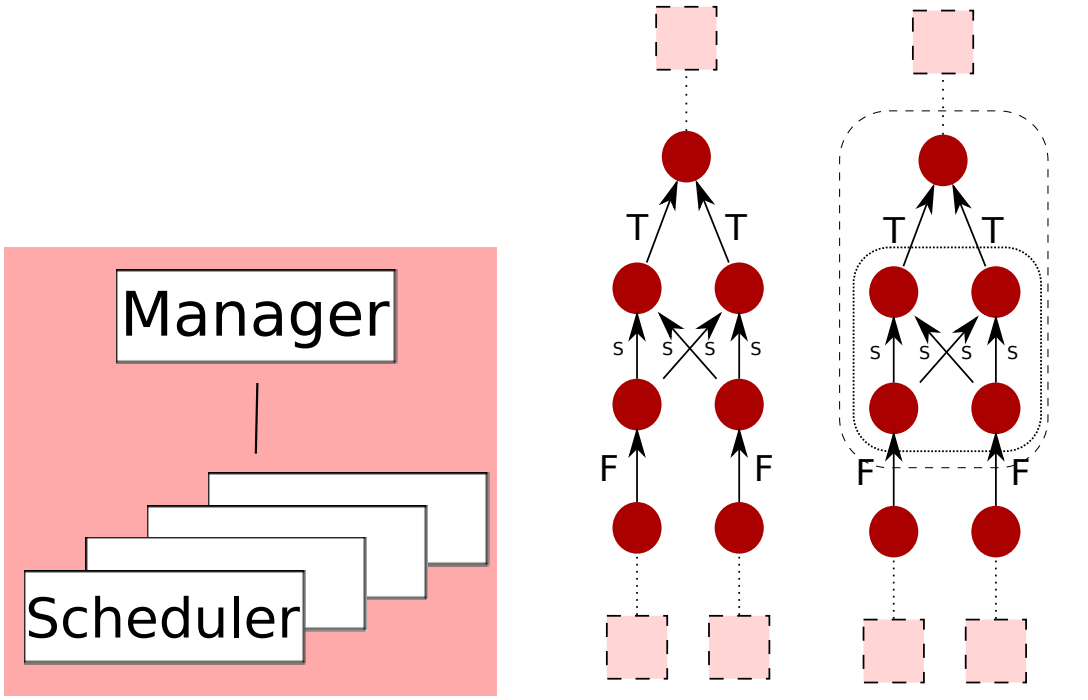
The optional multiplicity parameter, if omitted, means pairwise association: each member of `group1` has a communication channel to each member of `group2`. If present, say with value x, it specifies that each member of `group1` has x communication channels to members of `group2`, assigned through members of `group1` in a round-robin fashion.

3.4 Channel Elements

All the records transmitted through communication channels are encapsulated in serializable objects called `ChannelElement`s. By default, they have only a description field and a information field, but it can be extended by clients to support other operations. Moreover, client applications can decide whether they prefer coarse-grained encapsulation (big records), fine-grained encapsulation (small records), or a mix of both. In any case, it is usually convenient that application developers extend the `ChannelElement` class to support application-specific operations on the packed data block (like insert/removal/search operations, metadata information handling, among others).

3.5 The Manager

The manager receives application specifications and creates one application handler for initiated application. Particularly, the handler contains one scheduler for application, executing in multiple threads. See Fig. 4a.



(a) Manager and schedulers. Each application has one scheduler, which identifies `NodeGroup`s and `NodeBundle`s.

(b) The inner dashed selection is a `NodeGroup`; the outer dashed selection is a `NodeBundle`.

Figure 4: The manager and one scheduler instance for each running application.

The scheduling process consists first on identifying groups nodes that communicate via shared memory in the application graph. Whenever two nodes are connected by a shared-memory communication channel, they are in the same `NodeGroup`, and this relation is transitive. In Fig. 4b, the inner dashed segment represents an example of these groups, which will be assigned simultaneously to a single launcher. These groups are called `NodeGroup`s.

The scheduler then identifies parts that, although possibly assigned to different launchers, should be scheduled simultaneously across many launchers. The logic to identify `NodeBundle`s is analogous to the previous scenario, but now the scheduler looks for nodes linked by TCP communication channels, also in

a transitive fashion. In Fig. 4b, the outer dashed segment represents an example of these parts, called `NodeBundle`s.

Each application scheduler keeps track of scheduled `NodeGroup`s for the corresponding application. When they are finished, their execution summary, including timing information for the whole `NodeGroup` and for each node that is part of it, is returned to the manager. When all the dependencies (all the input files) for a `NodeBundle` are already created, the `NodeBundle` is released and its `NodeGroup`s are scheduled to different launchers.

3.6 The Launchers

The launchers receive `NodeGroup`s from the manager and execute them. For each `NodeGroup` received, an `ExecutionHandler` is created. The execution handler is responsible for starting the client-specified nodes on the local machine. Among the tasks carried by the `ExecutionHandler` are:

1. Setting up the shared-memory pipes, starting TCP servers or creating output files, according to the communication channels involved;

2. Connecting nodes that communicate via shared-memory pipes, and mapping communication channel names and actual pipes at the output of the origin node and at the input of the target node – this allows the `read*()` and `write*()` functions to choose specific destinations and sources, respectively;

3. Notifying the manager about the address and port number of each started TCP server. A mapping between target nodes of a TCP communication channel and the associated server addresses is now maintained at the manager. This allows nodes with outgoing TCP channels to query the manager for the associated address of the target node, which resides in a different machine. Creating this mapping at the moment of launching nodes is needed because TCP addresses from TCP servers are assigned dynamically by the operating system.

4. Obtaining from the manager the addresses associated with outgoing TCP channels. This is the reverse step of the previous item, and is carried by `ExecutionHandler`s that execute nodes that have outgoing TCP connections.

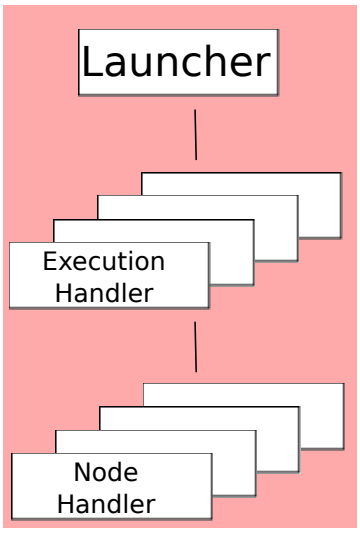


Figure 5: The launcher and one execution handler for each running NodeGroup. Each execution handler initiates one node handler for each contained Node.

For each node which is part of some `NodeGroup`, a special helper thread is created to run it. This helper thread, called `NodeHandler`, runs the node and records statistics for its execution, including the CPU/sys/user time spent (the CPU and sys can be obtained up to nanosecond granularity). When the

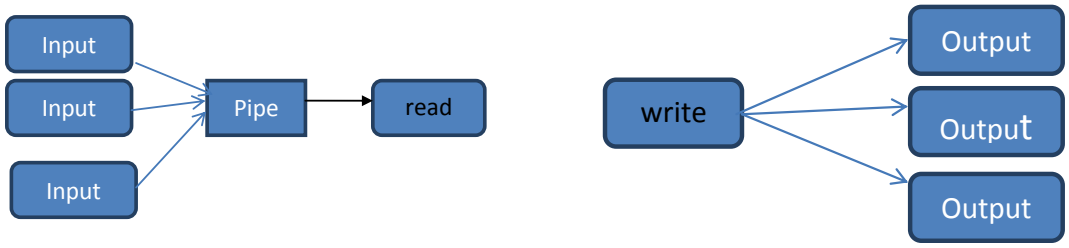
execution is finished, a result summary is prepared, containing all the collected statistics, and sent to the manager.

4 Running C/C++ Applications

As previously discussed, our system currently supports three types of communication. The choice of communication type greatly affects system behavior, and it is up to the users to decide which type is the best for their problem.

4.1 Implementation

The nodes will be started by the local launcher. The launcher will pass the parameters to our wrapper for all input and output connections through the command line. The node will be initialized and then it will wait to receive records. Each input connection is handled by a different thread. When a record is received by an input connection it is written to a pipe from where the application process reads. If the pipe is full the thread will block until there is enough space in the pipe. Output connections work similarly like input connections with the exception that records are not buffered through a pipe but sent immediately through the output streams



(a) Input implementation.

(b) Output implementation.

Figure 6: Implementation of the wrapper routines for C/C++ applications.

4.2 API

Our initial design was to support parallelization for existing C programs that use command line as input and output without any modification. The C wrapper we wrote for this was creating two named pipes in which file descriptor were duped as `stdin` and `stdout` and then the wrapper created a child process which call `execve()` to run the program. Unfortunately, we were forced to abandon this approach because we couldn't create a meaningful way to map the output. The output produced by the C program had to be send to all reducers without any distribution of the records. Our current implementation requires small modifications in order to run existing programs. In particular, instead of using command line as output and input the programs will need to use our communication API.

Our main goal in constructing the API was to give great flexibility to the users. They could select how the output of their program will be distributed. In order to achieve this goal and making it easy for user to send and receive records we created the following functions.

Function	Explanation
outputConnectionNames()	gets all output connection names
readLine(buffer,size)	reads a record (each records is treated as one line)
readMax(buffer,size)	read up to the specified size of the buffer
writeAll(buffer)	writes to all available connections
writeSpecific(buffer, connections, size)	writes to the specified connections
writeRandom(buffer)	Writes to a random connection

The API is created as a static library. In order to use it, a developer needs to include the library header file in their application. Secondly, during compilation the static library and pthread library should be linked. Lastly, the developer needs to specify DAG graph with our manager, so the program can be effectively parallelized.

5 Evaluation

In this section, we present tests to evaluate the system. Our tests consist on two micro-benchmarks and two complete MapReduce applications. The micro-benchmarks are described below.

Communication stress Two nodes, connected by a TCP communication channel, read and relay 118MB and 1.2GB-sized files containing very fine-grained `ChannelElements`, totaling 6 million and 60 million objects. The average `ChannelElement` size is 19.89 bytes. Common applications will usually employ much bigger records (in the order of MB), but we want to generate many records for this stress benchmark (we want to test the Java garbage collector). We verify the memory consumption of the executing launcher at different moments, as well as the CPU/user times devoted to the application.

Scheduling stress This micro-benchmark evaluates the performance of the graph parsing and scheduling routines at the manager. A client submits application specifications with 100 and 1000 nodes, with, respectively, about 500 and 50000 edges. The procedure to generate edges is the following: (1) we index the nodes, from n_1 to n_x, where x is the number of nodes considered; (2) each node n_i is connected to every other node $n_{j>i}$ with probability 0.1. The number of edges is therefore quadratic in the number of nodes. The tasks are almost trivial (they just establish communication channels and then exit). All the communication channels are TCP-based. This choice for TCP channels is based by the fact that TCP-based communication requires *more* interaction between the launchers and the manager, because the mapping between target node names and server addresses is handled by the manager. We analyze the total time to schedule the *NodeGroups*.

The micro-benchmarks are performed on an Core i5 with two cores (with hyper-threading technology enabled, which simulates four cores). The system has 4GB of DDR3 RAM, and runs MacOS X 10.6.5 (10H574). The hard disk is a 320GB Hitachi model HTS545032B9SA02.

For the first micro-benchmark, we generate the input files as described above. The results are presented in Tab. 2. We see that the memory consumption kept steady as the number of manipulated `ChannelElement`s increased, showing that the Java runtime system reclaimed unused memory in an expected manner, without big impacts in the application footprint. If this were not true, nodes processing huge files might exhaust memory. Moreover, Tab. 2 shows a linear increase on the CPU and user times for the communication stress benchmark.

Input size	# of `ChannelElements`	Avg. CPU time	Avg. user time	Memory consumption (MB)
118MB	6,220,100	2.18 min	0.76 min	130.6
1.2GB	62,201,000	21.30 min	7.10 min	136.6

Table 2: Communication stress benchmark results.

Communication stress benchmarks were important because they allowed us to identify an interesting bug in the java `ObjectOutputStream` class: as we kept writing objects to it, it kept consuming more and more memory, as the class *stores* a copy of sent objects. Left untreated, this eventually consumes all the available memory. Fortunately, there is a method that resets the output device, cleaning the unused memory. We call it once every 64K transmitted objects.

For the second micro-benchmark, we generate the graphs, according to the previous description, and submit the application. We are interested on the time to parse the graph (identifying `NodeGroup`s and `NodeBundle`s). The results are presented on Tab. 3.

# of nodes	# of edges	Time to parse graph
100	502	19 msec
1,000	50,127	604 msec

Table 3: Scheduling stress benchmark results.

Finally, we ran two MapReduce applications consisting on counting the words of 3.7GB and 37GB of data, split in 31 input files, and sorting the words by the number of occurrences. As in the case of the communication benchmark, to stress communication, we used very fine-grained records: in total we processed 192,823,100 and 1,928,231,000 `ChannelElement`s. The setting is as Fig. 7. We employed 31 mappers and 31 reducers, executed across the department new cluster[1]. The launchers were submitted via Sun GridEngine to the cluster, and they registered with the manager running on machine `barney`. Each machine executed two launchers, as they were dual-core machines (except for one case, which executed one launcher). The machines were dedicated to running our launchers.

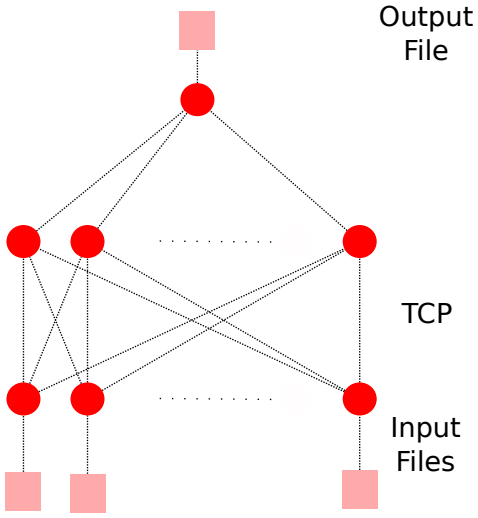


Figure 7: Our evaluation scenario. In both tests, the number of mappers and reducers is 31. In the first test, the input files have 3.7GB, and contain 192,823,100 `ChannelElement`s in total. In the second test, the input files have 37GB, and contain 1,928,231,000 `ChannelElements` in total.

Our MapReduce applications employed a combiner function at the mappers to avoid transmitting too much data to the reducers. Therefore, the communication between mappers and reducers is not our main concern. Tab. 4 shows the results.

[1]Thanks to John Bazik for the support.

Data size	Number of records	Total time	Avg. node CPU time	Avg. node user time
3.7 GB	192,823,100	14min 25 sec	57 sec	30 sec
37 GB	1,928,231,000	136min 14 sec	9min 49sec	5 min 36 sec

Table 4: MapReduce experimental results.

Both tests show big differences in the total real time vs average CPU/user time. These discrepancies occur in the mappers, and we believe that, for most of the time, the mappers were waiting for I/O operations to complete in the department filesystem[2]. In Fig. 8 and Fig. 9 we present graphically the CPU and user times for all the mappers in both tests.

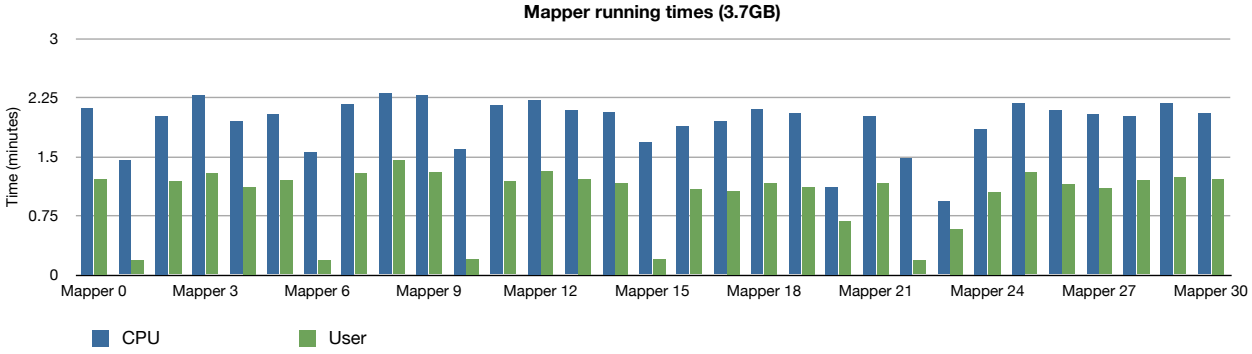


Figure 8: CPU and user times for mappers in the 3.7GB evaluation.

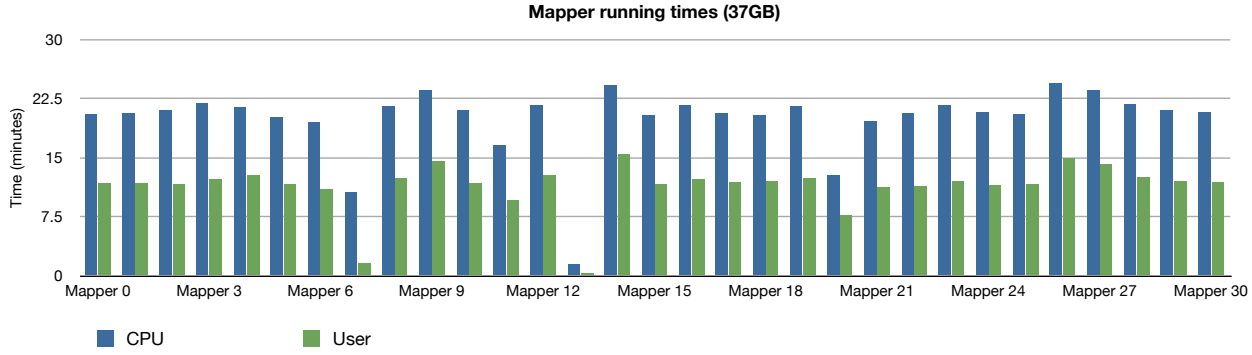


Figure 9: CPU and user times for mappers in the 37GB evaluation.

In Fig. 10 and Fig. 11 we present graphically the CPU and user times for all the reducers in both tests. We see that reducers were much faster than mappers. This is due to the low amount of data they actually manipulate, thanks to the combiner function employed by the mappers.

We also note that there is a higher variance in execution time for the reducers. This happens because the amount of pairs `<mapped-value,data-record>` transmitted from the mappers vary across the reducers. We used a simple hash function taken over the `ChannelElement`'s contents to decide to which reducer a pair `<mapped-value,data-record>` should be sent to.

[2]The tests were run on Dec 14, 2010, between 3PM and 7PM.

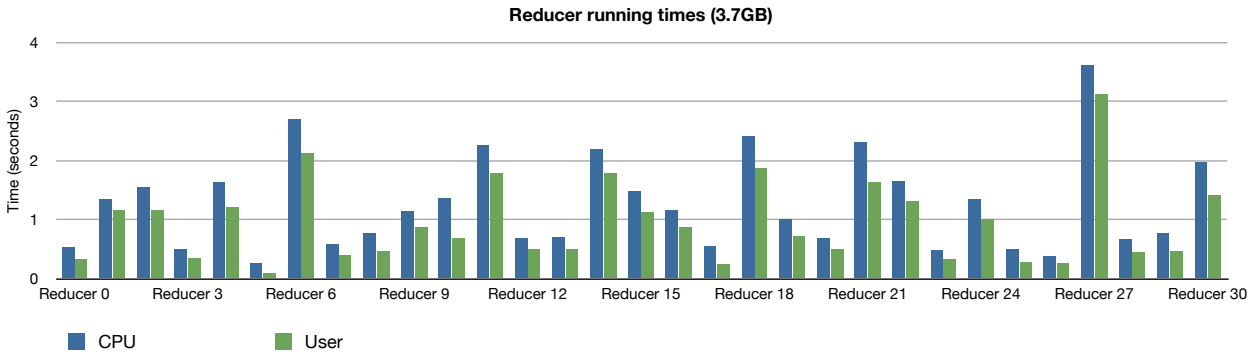


Figure 10: CPU and user times for reducers in the 3.7GB evaluation.

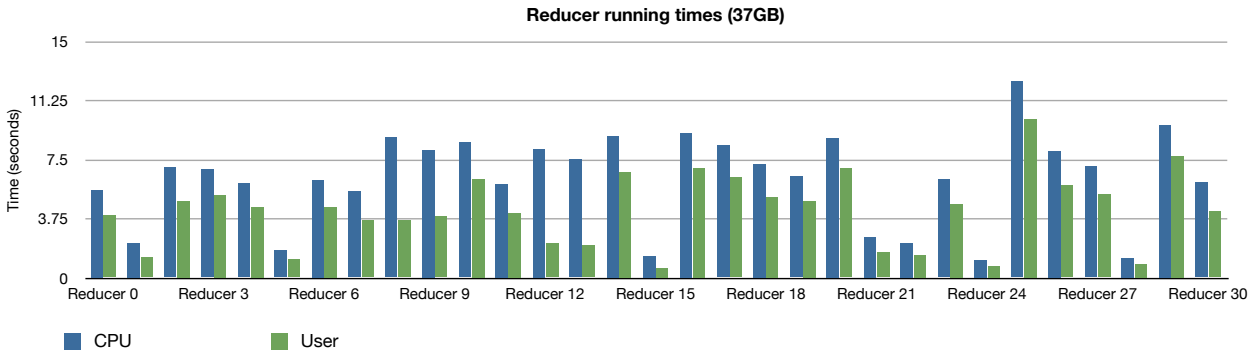


Figure 11: CPU and user times for reducers in the 37GB evaluation.

6 Conclusion

We implemented a flexible distributed runtime system that provides a foundation for future research investigation and prototyping. All the implemented functionality was tested in synchrony with the development process. This report presents a usage scenario that, through restricted, broadly encompasses system features:

- Restricted communication structure - we only evaluated MapReduce applications, although the system is capable of handling different communication topologies. However, we believe that testing our MapReduce abstraction is sufficient *for now* because:
  1. It uses the main underlying system features (scheduling, staging, and communication routines) as any general application;
  2. It allows us testing how expressive and convenient the MapReduce abstraction really is (these features are system goals, and should be evaluated too).
- Restricted scale – our system was tested in a relatively small scale compared to a real application scenario. However, we believe that this test was big enough to permit us to identify (and to allow us to correct) the most evident/serious scaling problems, the ones that would probably make bigger scale setups impossible.

It is important to mention features for runtime systems that were only partially implemented or not implemented at all with this work, but nevertheless are important in real deployments:

Fault Tolerance. If the manager detects a launcher stopped at the moment of using it, it removes such launcher from its registered launchers list. The launcher should register again with the manager to be considered again for remote execution. However, if a launcher fails while nodes are being executed, the result summary never returns, which might block `NodeBundle`s to be released.

Local File Support. We only support files present in a distributed filesystem, and accessible to all the nodes. However, it is important to support local files whenever a distributed filesystem is not available.

Locality Features. If we supported local files, it would be convenient that the `NodeGroup` scheduling tried to accommodate those entities on machines that most probably have the needed files. It would also be interesting to support *in-memory* record storage on the launchers to avoid hitting the disk every time an affine `NodeGroup` requests the information.

Accounting. This feature was implemented in regard to CPU timing, but it would be interesting to implement network accounting. Counting the number of `ChannelObject`s transmitted requires a simple changes to the system, but counting the number of *bytes* sent/received requires more work.

In the context of the course, we believe that this project enabled us to expand our perception regarding the complexity involved in distributed applications, particularly runtime systems. Many details that are usually overlooked in a paper description were evident in the design of the system.

In conclusion, we believe that the system provides the foundational features that we need in order to proceed with further research on distributed runtime systems. Moreover, its design and implementation amplified our understanding on the subject, an expected and useful outcome of this work.

References

[1] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *NSDI'10: Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[3] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[4] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency and Computation : Practice and Experience*, 17:323–356, February 2005.