Nephele: Efficient Parallel Data Processing in the Cloud

Daniel Warneke
Technische Universität Berlin
Berlin, Germany
daniel.warneke@tu-berlin.de

Odej Kao
Technische Universität Berlin
Berlin, Germany
odej.kao@tu-berlin.de

ABSTRACT

In recent years Cloud Computing has emerged as a promising new approach for ad-hoc parallel data processing. Major cloud computing companies have started to integrate frameworks for parallel data processing in their product portfolio, making it easy for customers to access these services and to deploy their programs. However, the processing frameworks which are currently used stem from the field of cluster computing and disregard the particular nature of a cloud. As a result, the allocated compute resources may be inadequate for big parts of the submitted job and unnecessarily increase processing time and cost. In this paper we discuss the opportunities and challenges for efficient parallel data processing in clouds and present our ongoing research project Nephele. Nephele is the first data processing framework to explicitly exploit the dynamic resource allocation offered by today's compute clouds for both, task scheduling and execution. It allows assigning the particular tasks of a processing job to different types of virtual machines and takes care of their instantiation and termination during the job execution. Based on this new framework, we perform evaluations on a compute cloud system and compare the results to the existing data processing framework Hadoop.

Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming—*Distributed programming*

General Terms

Performance, Design, Economics

Keywords

Many-Task Computing, High-Throughput Computing, Loosely Coupled Applications, Cloud Computing

1. INTRODUCTION

Today a growing number of companies have to process huge amounts of data in a cost-efficient manner. Classical representatives for these companies are operators of Internet search engines, like Google, Yahoo or Microsoft. The vast amount of data they have to deal with every day has made traditional database solutions prohibitively expensive [5]. Instead, these companies have popularized an architectural paradigm that is based on a large number of shared-nothing commodity servers. Problems like processing crawled documents, analyzing log files or regenerating a web index are split into several independent subtasks, distributed among the available nodes and computed in parallel.

In order to simplify the development of distributed applications on top of such architectures, many of these companies have also built customized data processing frameworks in recent years. Examples are Google's MapReduce engine [9], Microsoft's Dryad [13], or Yahoo!'s Map-Reduce-Merge [6]. They can be classified by terms like high throughput computing (HTC) or many-task computing (MTC), depending on the amount of data and the number of tasks involved in the computation [19]. Although these systems differ in design, the programming models they provide share similar objectives, namely hiding the hassle of parallel programming, fault tolerance and execution optimizations from the developer. Developers can typically continue to write sequential programs. The processing framework then takes care of distributing the program among the available nodes and executes each instance of the program on the appropriate fragment of data.

For companies that only have to process large amounts of data on an irregular basis running their own data center is obviously not an option. Instead, *cloud computing* has emerged as a promising approach to rent a large IT infrastructure on a short-term pay-per-usage basis. Operators of compute clouds, like Amazon EC2 [1], let their customers allocate, access and control a set of virtual machines which run inside their data centers and only charge them for the period of time the machines are allocated. The virtual machines are typically offered in different types, each type with its own characteristics (number of CPU-cores, amount of main memory, etc.) and cost.

Since the virtual machine abstraction of compute clouds fits the architectural paradigm assumed by the data processing frameworks described above, projects like Hadoop [23], a popular open source implementation of Google's MapReduce framework, already began to promote using their frameworks in the cloud [25]. Only recently, Amazon has integrated Hadoop as one of its core infrastructure services [2]. However, instead of embracing its dynamic resource allocation, current data processing frameworks rather expect the cloud to imitate the static nature of the cluster environments they were originally designed for. E.g., at the moment the types and number of virtual machines allocated at the beginning of a compute job cannot be changed in the course of processing, although the tasks the job consists of might have completely different demands on the environment. As a result, rented resources may be inadequate for big parts of the processing job, which may lower the overall processing performance and increase the cost.

In this paper we want to discuss the particular challenges and opportunities for efficient parallel data processing inside clouds and present *Nephele*, a new processing framework explicitly designed for cloud environments. Most notably, Nephele is the first data processing framework to include the possibility of dynamically allocating/deallocating different compute resources from a cloud in its scheduling and during job execution.

The rest of this paper is structured as follows: Section 2 starts with analyzing the above mentioned opportunities and challenges and derives some important design principles for our new framework. In Section 3 we present Nephele's basic architecture and outline how jobs can be described and executed inside the cloud. Section 4 provides some first figures on Nephele's performance and the impact of the optimizations we propose. Finally, our work is concluded by related work (Section 5) and ideas for future work (Section 6).

2. CHALLENGES AND OPPORTUNITIES

Current data processing frameworks like Google's MapReduce or Microsoft's Dryad engine have been designed for cluster environments. This is reflected in a number of assumptions they make which are not necessarily valid in cloud environments. In this section we discuss how abandoning these assumptions raises new opportunities but also challenges for efficient parallel data processing in clouds.

2.1 Opportunities

Today's processing frameworks typically assume the resources they manage to consist of a *static* set of *homogeneous* compute nodes. Although designed to deal with individual nodes failures, they consider the number of available machines to be constant, especially when scheduling the processing job's execution. While compute clouds can certainly be used to create such cluster-like environments, much of their flexibility remains unused.

One of a compute cloud's key features is the provisioning of compute resources on demand. New virtual machines can be allocated at any time through a well-defined interface and become available in a matter of seconds. Machines which are no longer used can be terminated instantly and the cloud customer will be charged for them no more. Moreover, cloud operators like Amazon let their customers rent virtual machines of different types, i.e. with different computational power, different sizes of main memory and storage. Hence, the compute resources available in a cloud are highly *dynamic* and possibly *heterogeneous*.

With respect to parallel data processing, this flexibility leads to a variety of new possibilities, particularly for scheduling data processing jobs. The question a scheduler has to answer is no longer "Given a set of compute resources, how to distribute the particular tasks of a job among them?", but rather "Given a job, what compute resources match the tasks the job consists of best?". This new paradigm allows allocating compute resources dynamically and just for the time they are required in the processing work flow. E.g., a framework exploiting the possibilities of a cloud could start with a single virtual machine which analyzes an incoming job and then advises the cloud to directly start the required virtual machines according to the job's processing phases. After each phase, the machines could be released and no longer contribute to the overall cost for the processing job. More advanced processing frameworks could also take up the idea of Service Level Agreements (SLAs) [4] which guarantee that the execution of a processing job can be finished within a certain amount of time and at a certain cost.

Facilitating such use cases imposes some requirements on the design of a processing framework and the way its jobs are described. First, the scheduler of such a framework must become aware of the cloud environment a job should be executed in. It must know about the different types of available virtual machines as well as their cost and be able to allocate or destroy them on behalf of the cloud customer.

Second, the paradigm used to describe jobs must be powerful enough to express dependencies between the different tasks the jobs consists of. The system must be aware of which task's output is required as another task's input. Otherwise the scheduler of the processing framework cannot decide at what point in time a particular virtual machine is no longer needed to complete the overall job and deallocate it. The MapReduce pattern is a good example of an unsuitable paradigm here: Although at the end of a job only few reducer tasks may still be running, it is not possible to shut down the idle virtual machines, since it is unclear whether they contain intermediate results which are still required.

Finally, the scheduler of such a processing framework must be able to determine which task of a job should be executed on which type of virtual machine and, possibly, how many of those. This information could be either provided externally, e.g. as an annotation to the job description, or deduced internally, e.g. from collected statistics, similarly to the way database systems try to optimize their execution schedule over time [22].

2.2 Challenges

The cloud's virtualized nature helps to enable promising new use cases for efficient parallel data processing. However, it also imposes some new challenges compared to classical cluster setups. The major challenge we see is the *opaqueness* of clouds with prospect to exploiting data locality:

In a cluster the compute nodes are typically interconnected through a physical high-performance network. The topology of the network, i.e. the way the compute nodes are physically wired to each other, is usually well-known and, what is more important, does not change over time. Current data processing frameworks offer to leverage this knowledge about the network hierarchy and attempt to schedule tasks on compute nodes so that data sent from one node to the other has to traverse as few network switches as possible [9]. That way network bottlenecks can be avoided and the overall throughput of the cluster can be improved.

In a cloud this topology information is typically not exposed to the customer [25]. Since the nodes involved in processing a data intensive job often have to transfer tremendous amounts of data through the network, this drawback is particularly severe; parts of the network may become congested while others are essentially unutilized. Although there has been research on inferring likely network topologies solely from end-to-end measurements (e.g. [7]), it is unclear if these techniques are applicable to compute clouds. For security reasons clouds often incorporate network virtualization techniques (e.g. [8]) which can hamper the inference process, in particular when based on latency measurements.

Even if it was possible to determine the underlying network hierarchy inside a cloud and use it for topology-aware scheduling, the obtained information does not necessarily remain valid for the entire processing time. Virtual machines may be migrated for administrative purposes between different locations inside a cloud operator's data center without any notification, rendering any previous knowledge of the relevant network infrastructure obsolete.

As a result, the only way to ensure locality between tasks of a processing job is currently to execute these tasks on the same virtual machine in the cloud. This may involve allocating fewer, but more powerful virtual machines with multiple CPU cores. E.g., consider an aggregation task receiving data from seven generator tasks. Data locality can be ensured by scheduling these tasks to run on a virtual machine with eight cores instead of eight distinct single-core machines. However, currently no data processing framework we are aware of includes such strategies in its scheduling algorithms.

3. DESIGN

Based on the challenges and opportunities outlined in the previous section we have designed *Nephele*, a new data processing framework for cloud environments. Nephele takes up many ideas of previous processing frameworks but refines them to better match the dynamic and opaque nature of a cloud.

3.1 Architecture

Nephele's architecture follows a classical master-worker pattern as illustrated in Fig. 1.

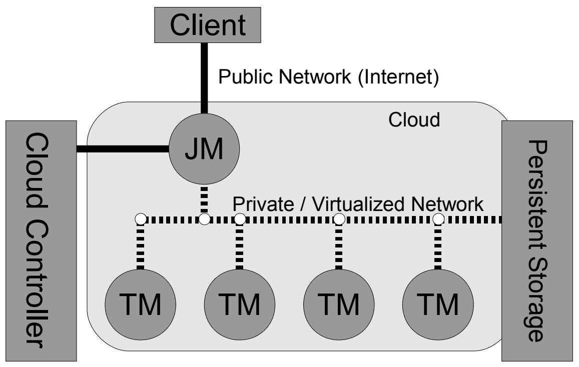

Figure 1: Structural overview of Nephele running inside a compute cloud

Before submitting a Nephele compute job, a user must start an instance inside the cloud which runs the so called *Job Manager* (JM). The Job Manager receives the client's jobs, is responsible for scheduling them and coordinates their execution. It is capable of communicating with the cloud controller through a web service interface and can allocate or deallocate virtual machines according to the current job execution phase. We will comply with common cloud computing terminology and refer to these virtual machines as *instances* for the remainder of this paper. The term *instance type* will be used to differentiate between virtual machines with different hardware characteristics.

The actual execution of tasks which a Nephele job consists of is carried out by a set of instances. Each instance runs a local component of the Nephele framework we call a *Task Manager* (TM). A Task Manager receives one or more tasks from the Job Manager at a time, executes them and after that informs the Job Manager about their completion or possible errors. Unless a job is submitted to the Job Manager, we expect the set of instances (and hence the set of Task Managers) to be empty. Upon job reception the Job Manager then decides, depending on the particular tasks inside the job, how many and what type of instances the job should be executed on, and when the respective instances must be allocated/deallocated in order to ensure a continuous but cost-efficient processing. The concrete strategies for these scheduling decisions are explained later in this section.

The newly allocated instances boot up with a previously compiled virtual machine image. The image is configured to automatically start a Task Manager and register it with the Job Manager. Once all the necessary Task Managers have successfully contacted the Job Manager, it triggers the execution of the scheduled job.

Initially, the virtual machines images used to boot up the Task Managers are blank and do not contain any of the data the Nephele job is supposed to operate on. As a result, we expect the cloud to offer persistent storage (like e.g. Amazon S3 [3]). This persistent storage is supposed to store the job's input data and eventually receive its output data. It must be accessible for both the Job Manager as well as for the set of Task Managers, even if they are connected by a private or virtual network.

3.2 Job description

Similar to Microsoft's Dryad [13], jobs in Nephele are expressed as a directed acyclic graph (DAG). Each vertex in the graph represents a task of the overall processing job, the graph's edges define the communication flow between these tasks. We also decided to use DAGs to describe processing jobs for two major reasons:

The first reason is that DAGs allow tasks to have multiple input and multiple output edges. This tremendously simplifies the implementation of classical data combining functions like, e.g., join operations [6]. Second and more importantly, though, the DAG's edges explicitly model the communication paths which exist inside the processing job. As long as the particular processing tasks only exchange data through these designated communication edges, Nephele can always keep track of what instance might still require data from what other instances and which instance can potentially be shut down and deallocated.

Defining a Nephele job comprises three mandatory steps: First, the user must write the program code for each task of his processing job or select it from an external library. Second, the task program must be assigned to a vertex. Finally, the vertices must be connected by edges to define the communication paths of the job.

Tasks are expected to contain sequential code and process so-called *records*, the primary data unit in Nephele. Users may define arbitrary types of records, all implementing a common interface. From a programmer's perspective records enter and leave the task program through input or output gates. A task may have an arbitrary number of these input and output gates, which are at runtime connected by the Nephele framework to transport records from one task to the other.

After having specified the code for the particular tasks of the job, the user must define a so-called *Job Graph*. The Job Graph maps each task to a vertex of a directed acyclic graph (DAG) and determines the communication paths between these. Vertices with either no incoming or outgoing edges are treated specially in Nephele: The tasks assigned to these vertices are considered to be either data sources (input vertices) or sinks (output vertices) in the processing workflow. They can be associated with a URL pointing to where to read or write the data. Figure 2 illustrates the simplest possible Job Graph, consisting only of one input, one task and one output vertex.

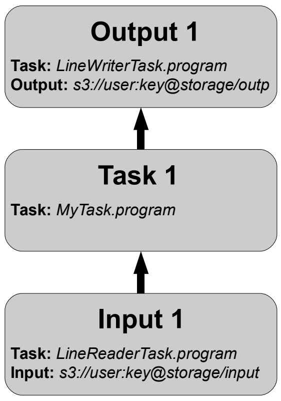
Output 1
Task: LineWriterTask.program
Output: s3://user:key@storage/outp
Task 1
Task: MyTask.program
Input 1
Task: LineReaderTask.program
Input: s3://user:key@storage/input

Figure 2: An example of a Job Graph in Nephele

One major design goal of Job Graphs has been simplicity: Users should be able to describe tasks and their relationships on a very abstract level, leaving aspects like task parallelization and the mapping to instances to Nephele. However, users who wish to specify these aspects explicitly can provide further annotations to their job description. These annotations include:

- **Number of subtasks:** A developer can declare his task to be suitable for parallelization. Users that include such tasks in their Job Graph can specify how many parallel *subtasks* Nephele should split the respective task into at runtime. Subtasks execute the same task code, however, they typically process different fragments of the data.

- **Number of subtasks per instance:** By default each (sub)task is assigned to a separate instance. In case several (sub)tasks are supposed to share the same instance, the user can provide a corresponding annotation with the respective task.

- **Sharing instances between tasks:** Subtasks of different tasks are usually assigned to different (sets of) instances unless prevented by another scheduling restriction. If a set of instances should be shared between different tasks the user can attach a corresponding annotation to the Job Graph.

- **Channel types:** For each edge connecting two vertices the user can determine a so-called channel type. Before executing a job, Nephele requires all edges of the original Job Graph to be replaced by a channel type. The channel type specifies how records are transported from one (sub)task to another at runtime. The choice of the channel type can have several implications on the entire job schedule including when and on what instance a (sub)task is executed.

- **Instance type:** A (sub)task can be executed on different instance types which may be more or less suitable for the considered program. Therefore we have developed special annotations task developers can use to characterize the hardware requirements of their code. However, a user who simply utilizes these annotated tasks can also overwrite the developer's suggestion and explicitly specify the instance type for a task in the Job Graph.

If the user omits to augment the Job Graph with these specifications, Nephele's scheduler will apply default strategies which are discussed in the following subsection.

Once the Job Graph is specified, the user can submit it to the Job Manager, together with the credentials he obtained from his cloud operator. The credentials are required since the Job Manager must allocate/deallocate instances from the cloud during the job execution on behalf of the user.

3.3 Job Scheduling and Execution

After having received a valid Job Graph from the user, Nephele's Job Manager transforms it into a so-called *Execution Graph*. An Execution Graph is Nephele's primary data structure for scheduling and monitoring the execution of a Nephele job. Unlike the abstract Job Graph, the Execution Graph contains all the concrete information required to schedule and execute the received job on the cloud. Depending on the level of annotations the user has provided with his Job Graph, Nephele may have different degrees of freedom in constructing the Execution Graph. Figure 3 shows one possible Execution Graph constructed from the previously depicted Job Graph (Figure 2).

In contrast to the Job Graph, an Execution Graph is no longer a pure DAG. Instead, its structure resembles a graph with two different levels of details, an abstract and a concrete level. While the abstract graph describes the job execution on a task level (without parallelization) and the scheduling of instance allocation/deallocation, the concrete, more finegrained graph defines the mapping of subtasks to instances and the communication channels between them.

On the abstract level, the Execution Graph equals the user's Job Graph. For every vertex of the original Job Graph there exists a so-called *Group Vertex* in the Execution Graph. As a result, Group Vertices also represent distinct tasks of the overall job, however, they cannot be seen as executable units. They are used as a management abstraction to control the set of subtasks the respective task program is split into. The edges between Group Vertices are

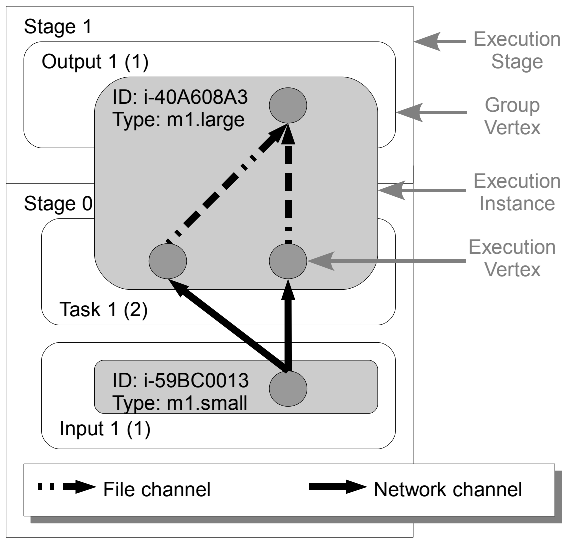

Figure 3: An Execution Graph created from the original Job Graph

only modeled implicitly as they do not represent any physical communication paths during the job processing. For the sake of presentation, they are also omitted in Figure 3.

In order to ensure cost-efficient execution on a compute cloud, Nephele allows to allocate instances in the course of the processing job, when some (sub)tasks have already been completed or are already running. However, this just-in-time allocation can also cause problems, since there is the risk that the requested instance types are temporarily not available in the cloud. To cope with this problem, Nephele separates the Execution Graph into one or more so-called *Execution Stages*. An Execution Stage must contain at least one Group Vertex. Its processing can only start when all of its preceding stages (i.e. the subtasks included in the respective Group Vertices) have been successfully processed. Based on this Nephele's scheduler ensures the following three properties for the entire job execution: First, when the processing of a stage begins, all instances required within the stage are allocated. Second, all subtasks included in this stage are set up (i.e. sent to the corresponding Task Managers along with their required libraries) and ready to receive records. Third, before the processing of a new stage, all intermediate results of its preceding stages are stored in a persistent manner. Hence, Execution Stages can be compared to checkpoints. In case a sufficient number of resources cannot be allocated for the next stage, they allow a running job to be interrupted and later on restored when enough spare resources have become available.

The concrete level of the Execution Graph refines the job schedule to include subtasks and their communication channels. In Nephele, every task is transformed into either exactly one, or, if the task is suitable for parallel execution, at least one subtask. For a task to complete successfully, each of its subtasks must be successfully processed by a Task Manager. Subtasks are represented by so-called *Execution Vertices* in the Execution Graph. They can be considered the most fine-grained executable job unit. To simplify management, each Execution Vertex is always controlled by its corresponding Group Vertex.

Nephele allows each task to be executed on its own instance type, so the characteristics of the requested virtual machines can be adapted to the demands of the current processing phase. To reflect this relation in the Execution Graph, each subtask must be mapped to a so-called *Execution Instance*. An Execution Instance is defined by an ID and an instance type representing the hardware characteristics of the corresponding virtual machine. It is a scheduling stub that determines which subtasks have to run on what instance (type). We expect a list of available instance types together with their cost per time unit to be accessible for Nephele's scheduler and instance types to be referable by simple identifier strings like "m1.small".

Before beginning to process a new Execution Stage, the scheduler collects all Execution Instances from that stage and tries to replace them with matching cloud instances. If all required instances could be allocated the subtasks are sent to the corresponding instances and set up for execution.

On the concrete level, the Execution Graph inherits the edges from the abstract level, i.e. edges between Group Vertices are translated into edges between Execution Vertices. In case of parallel task execution, when a Group Vertex contains more than one Execution Vertex, the developer of the consuming task can implement an interface which determines how to connect the two different groups of sub tasks.

Nephele requires all edges of an Execution Graph to be replaced by a channel before processing can begin. The type of the channel determines how records are transported from one subtask to the other. Currently, Nephele features three different types of channels, which all put different constrains on the Execution Graph.

- **Network channels:** A network channel lets two subtasks exchange data via a TCP connection. Network channels allow pipelined processing, so the records emitted by the producing subtask are immediately transported to the consuming subtask. As a result, two subtasks connected via a network channel may be executed on different instances. However, since they must be executed at the same time, they are required to run in the same Execution Stage.

- **In-Memory channels:** Similar to a network channel, an in-memory channel also enables pipelined processing. However, instead of using a TCP connection, the respective subtasks exchange data using the instance's main memory. Transporting records through in-memory channels is typically the fastest way to transport records in Nephele, however, it also implies the most scheduling restrictions: The two connected subtasks must be scheduled to run on the same instance and run and in the same Execution Stage.

- **File channels:** A file channel allows two subtasks to exchange records via the local file system. The records of the producing task are first entirely written to an intermediate file and afterwards read into the consuming subtask. Nephele requires two such subtasks to be assigned to the same instance. Moreover, the consuming group vertex must be scheduled to run in a higher Execution Stage than the producing group vertex. In general, Nephele only allows subtasks to exchange records across different stages via file channels because they are the only channel types which store the intermediate records in a persistent manner.

As mentioned above, constructing an Execution Graph from a user's submitted Job Graph may leave different degrees of freedom to Nephele. Using this freedom to construct the most efficient Execution Graph (in terms of processing time or monetary cost) is currently a major focus of our research. At the current state, we pursue a simple default strategy: Unless the user provides any job annotation which contains more specific instructions for the respective task, we create exactly one Execution Vertex for each vertex of the Job Graph. The default channel types are network channels. Each Execution Vertex is by default assigned to its own Execution Instance unless the user's annotations or other scheduling restrictions (e.g. the usage of in-memory channels) prohibit it. The default instance type to be used is the instance type with the lowest price per time unit available in the compute cloud.

In order to reflect the fact that most cloud providers charge their customers for the usage of instances by the hour, we integrated the possibility to reuse instances. Nephele can keep track of the instances' allocation times. An instance of a particular type which has become obsolete in the current Execution Stage is not immediately deallocated if an instance of the same type is required in an upcoming Execution Stage. Instead, Nephele keeps the instance allocated until the end of its current lease period. If the next Execution Stage has begun before the end of that period, it is reassigned to an Execution Vertex of that stage, otherwise it deallocated early enough not to cause any additional cost.

4. EVALUATION

In this section we want to present first performance results of Nephele and compare them to the data processing framework Hadoop. We have chosen Hadoop as our competitor, because it is open source software and currently enjoys high popularity in the data processing community. We are aware that Hadoop has been designed to run on a very large number of nodes (i.e. several thousand nodes). However, according to our observations, the software is typically used with significantly less instances in current compute clouds. In fact, Amazon itself limits the number of available instances for their MapReduce service to 20 unless the respective customer passes an extended registration process [2].

The challenge both frameworks are measured by consists of two abstract tasks: Given a set of random integer numbers, the first task is to determine the k smallest of those numbers. The second task subsequently is to calculate the average of those k smallest numbers. The job is a classical representative for a variety of data analysis jobs whose particular tasks vary in their demand of hardware environments. While the first task has to sort the entire data set and therefore can take advantage of large amounts of main memory and parallel execution, the second aggregation task requires almost no main memory and, at least eventually, cannot be parallelized. For both the Hadoop and the Nephele experiment, we chose the data set size to be 100 GB. Each integer number had the size of 100 bytes. As a result, the data set contained about 10^9 distinct integer numbers. k has been set to $2 \cdot 10^8$, so the smallest 20 % of all numbers had to be determined and aggregated.

4.1 General hardware setup

Both the Nephele as well as the Hadoop job were executed on our local compute cloud of commodity servers. Each server is equipped with a Intel Xeon 2.66 GHz CPU (8 CPU cores) and a total main memory of 32 GB. All servers are connected through regular 1 GBit/s Ethernet links. As host operating system we installed Gentoo Linux (kernel version 2.6.30) with KVM [14] (version 88-r1) and used virtio [21] to provide virtual I/O access.

For managing the cloud and provisioning virtual machines on request of Nephele, we set up Eucalyptus [15]. Similar to Amazon EC2, Eucalyptus offers a predefined set of instance types a user can choose from. During our experiment we used two different instance types: The first instance type is "m1.small" which corresponds to an instance with one CPU core, one GB of RAM and a 128 GB disk. The second instance type, "c1.xlarge", represents an instance with 8 CPU cores, 18 GB RAM and a 512 GB disk. Amazon EC2 defines comparable instance types and offers them at a price of 0.10 \$ or 0.80 \$ per hour, respectively.

The images used to boot up the instances contained a standard Ubuntu Linux (kernel version 2.6.28) with no additional software but a Java runtime environment, which is required by Nephele's Task Manager.

The 100 GB input data set of random integer numbers has been generated according to the rules of the Jim Gray sort benchmark [17]. In order to make the data accessible to Hadoop, we started an HDFS [23] data node on each of the allocated instances prior to the processing job and distributed the data evenly among the nodes. Since this initial setup procedure was necessary for both the Nephele and the Hadoop job, we have chosen to ignore it in the following performance discussion.

4.2 Setup for the Hadoop experiment

In order to execute the described task with Hadoop we created three different MapReduce jobs which are executed consecutively.

The first MapReduce job reads the entire input data set, sorts the contained integer numbers ascendingly and writes them back to Hadoop's HDFS file system. Since the MapReduce engine is internally designed to sort the incoming data between the map and the reduce phase, we did not have to provide custom map and reduce functions here. Instead, we simply used the Terasort code, which has recently been recognized for being well-suited for these kinds of tasks [17].

The second and third MapReduce jobs operate on the sorted data set and perform the data aggregation. Thereby, the second MapReduce job selects the output files from the preceding sort job which, just by their file size, must contain the smallest $2 \cdot 10^8$ numbers of the initial data set. The map function is fed with the selected files and emits the first $2 \cdot 10^8$ numbers to the reducer. In order to enable parallelization in the reduce phase, we chose the intermediate keys for the reducer randomly from a predefined set of keys. These keys ensure the emitted numbers are distributed evenly among the n reducers in the system. Each reducer then calculates the average of the received $\frac{2 \cdot 10^8}{n}$ integer numbers. The third MapReduce job finally reads the n intermediate average values and aggregates them to a single overall average.

Since Hadoop is not designed to deal with heterogeneous compute nodes we allocated six instances of type "c1.xlarge" for the experiment. We configured Hadoop to perform best for the first, computationally most expensive, MapReduce job: In accordance to [17] we set the number of map tasks per job to 48 (one map task per CPU core) and the number of reducers to 12. The memory heap of each map task as well as the in-memory file system have been increased to 1 GB and 512 MB, respectively, in order to avoid unnecessarily spilling transient data to disk.

4.3 Setup for the Nephele experiment

Figure 4 illustrates the Execution Graph we instructed Nephele to create in order to meet the processing challenge. For brevity, we omit a discussion on the original Job Graph. We pursue the overall idea that several powerful but expensive instances are used to determine the $2 \cdot 10^8$ smallest integer numbers in parallel while, after that, a single, inexpensive instance is utilized for the final aggregation. The graph contains five distinct tasks, split into several different groups of subtasks and assigned to different instance types.

The first task, *BigIntegerReader*, processes the assigned input files and emits each integer number as a separate record. The records are received by the second task, *BigIntegerSorter*, which attempts to buffer all incoming records into main memory. Once it has received all designated records, it performs an in-memory quick sort and subsequently continues to emit the records in an order-preserving manner. Since the BigIntegerSorter task requires large amounts of main memory we split it into 146 subtasks and assigned these evenly to six instances of type "c1.xlarge". The preceding BigIntegerReader task was also split into 146 subtasks and set up to emit records via in-memory channels.

The third task, *BigIntegerMerger*, receives records from multiple input channels. Once it has read a record from all available input channels, it sorts the records locally and emits them in ascending order. The BigIntegerMerger tasks occurs three times in a row in the Execution Graph. The first time it was split into six subtasks, one subtask assigned to each of the six "c1.xlarge" instances. As described in Section 2, this is currently the only way to ensure data locality between the sort and merge tasks. The second time the BigIntegerMerger task occurs in the Execution Graph, it is split into two subtasks. These two subtasks are assigned to two of the previously used "c1.xlarge" instances. The third occurrence of the task is assigned to new instance of the type "m1.small". Each of the merge subtasks was configured to stop execution after having emitted $2 \cdot 10^8$ records. The stop command is propagated to all preceding subtasks of the processing chain, which allows the Execution Stage to be interrupted as soon as the final merge subtask has emitted the $2 \cdot 10^8$ smallest records.

The fourth task, *BigIntegerAggregater*, reads the incoming records from its input channels and sums them up. It is also assigned to the single "m1.small" instance. Since we no longer require the six "c1.xlarge" instances to run once the final merge subtask has determined the $2 \cdot 10^8$ smallest numbers, we changed the communication channel between the final BigIntegerMerger and BigIntegerAggregater subtask to a file channel. That way the aggregation is pushed into the next Execution Stage and Nephele is able to deallocate the expensive instances.

Finally, the fifth task, *BigIntegerWriter*, eventually receives the calculated average of the $2 \cdot 10^8$ integer numbers and writes the value back to HDFS.

4.4 Results

Figure 5 and Figure 6 show the performance results of our Nephele and Hadoop experiment, respectively. Both plots illustrate the average instance utilization over time, i.e. the average utilization of all CPU cores in all the instances allocated for the job at the given point in time. The utilization of each instance has been monitored with the Unix command "top" and is broken down into the amount of time the CPU cores spent running the respective data processing framework (USR), the kernel and its processes (SYS), and the time waiting for I/O to complete. In order to illustrate the impact network communication, the plots additionally show the average amount of IP traffic flowing between the instances over time.

We begin with discussing the Nephele experiment first: At point (a) Nephele has successfully allocated all instances required to start the first Execution Stage. Initially, the BigIntegerReader subtasks begin to read their splits of the input data set and emit the created records to the BigIntegerSorter subtasks. At point (b) the first BigIntegerSorter subtasks switch from buffering the incoming records to sorting them. Here, the advantage of Nephele's ability to assign specific instance types to specific kinds of tasks becomes apparent: Since the entire sorting can be done in main memory, it only takes several seconds. Three minutes later (c), the first BigIntegerMerger subtasks start to receive the presorted records and transmit them along the processing chain.

Until the end of the sort phase Nephele can fully exploit the power of the six allocated "c1.xlarge" instance. After that period, for the merge phase, the computational power is no longer needed. From a cost perspective it is now desirable to deallocate the expensive instances as soon as possible. However, since they hold the presorted data sets, at least 20 GB of records must be transfered to the inexpensive "m1.small" instance first. Here we experienced the network to the bottleneck, so much computational power remains unused during that transfer phase. In general, this transfer penalty must be carefully considered when switching between different instance types in the course of the job execution. For the future we plan to integrate compression for file and network channels as a means to trade CPU against I/O load. Thereby, we hope to mitigate this drawback.

At point (d) the final BigIntegerMerger subtask has emitted the $2 \cdot 10^8$ smallest integer records to the file channel and advises all preceding subtasks in the processing chain to stop execution. All subtasks of the first stage have now been successfully completed. As a result, Nephele automatically deallocates the six instances of type "c1.xlarge" and continues the next Execution Stage with only one instance of type "m1.small" left. In that stage the BigIntegerAggregater subtask reads the $2 \cdot 10^8$ smallest integer records from the file channel and calculates the average of them. Since the six expensive "c1.xlarge" instances no longer contribute to the number of available CPU cores in that period, the processing power allocated from the cloud again fits the task to be completed. At point (e), after 33 minutes, Nephele has finished the entire processing job.

For the Hadoop job the resource utilization is comparable to the one of the Nephele job at the beginning: During the map (point (a) to (c)) and reduce phase (point (b) to (d)) of the first sort job, Figure 6 shows a fair instance utilization. For the following two MapReduce jobs, however, the allocated instances are oversized: The second job, whose map and reduce phases range from point (d) to (f) and point (e) to (g), respectively, can only utilize about one third of the available CPU capacity. The third job (running between

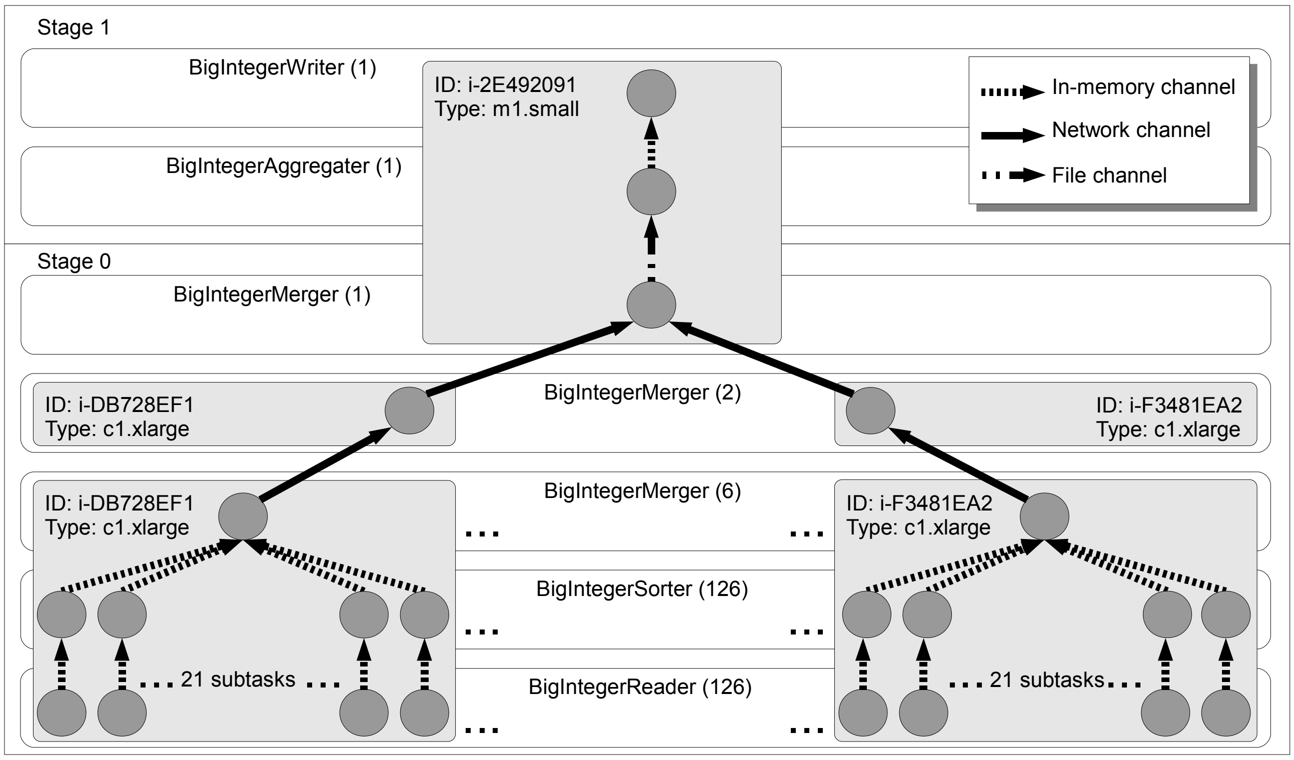

Figure 4: The Execution Graph for the Nephele experiment

point (g) and (h)) can only consume about 10 % of the overall resources.

The reason for Hadoop's eventual poor instance utilization is its assumption to run on a static compute cluster. In contrast to Nephele, it offers no notion of dynamic resource allocation/deallocation in its execution schedule. Once the MapReduce engine is started on a set of instances, no instance can be removed from that set without the risk of losing important intermediate results. As in this case, the result may be unnecessarily high processing costs.

Considering the short processing times of the presented tasks and the fact that most cloud providers offer to lease an instance for at least one hour, we are aware that Nephele's savings in time and cost might appear marginal at first glance. However, we want to point out that these savings grow by the size of the input data set. Due to the size of our test cloud we were forced to restrict data set size to 100 GB. For larger data sets, more complex processing jobs become feasible, which also promises more significant savings.

5. RELATED WORK

In recent years a variety of systems to facilitate the execution of loosely-coupled tasks on distributed systems has been developed. Although these systems typically share common overall goals (e.g. to hide issues of parallelism or fault tolerance from the user), they aim at different fields of application.

MapReduce [9] (or the open source version Hadoop [23]) is designed to run data analysis jobs on a large amount of data, which is expected to be stored across a large set of share-nothing commodity servers. MapReduce is highlighted by its simplicity: Once a user fit his program into the required map and reduce pattern, the execution framework takes care of splitting the job into subtasks, distributing and executing them. A single MapReduce job always consists of a distinct map and reduce program. However, several systems have been introduced to coordinate the execution of a sequence of MapReduce jobs with multiple tasks [18], [16].

MapReduce has been clearly designed for large static cluster environments. Although it can deal with sporadic node failures, the available compute nodes are essentially considered to be a fixed set of homogeneous machines.

The Pegasus framework by Deelman et al. [10] has been designed for mapping complex scientific workflows onto grid systems. Similar to Nepehle, Pegasus lets its users describe their jobs as a DAG with vertices representing the tasks to be processed and edges representing the dependencies between them. The created workflows remain abstract until Pegasus creates the mapping between the given tasks and the concrete compute resources available at runtime. The authors incorporate interesting aspects like the scheduling horizon which determines at what point in time a task of the overall processing job should apply for a compute resource. This is related to the stage concept in Nephele. However, Nephele's stage concept is designed to minimize the number allocated instances inside the cloud and clearly focuses on reducing cost. In contrast, Pegasus' scheduling horizon is used to deal with unexpected changes in the execution environment. Pegasus uses DAGMan and Condor-G [12] as its execution engine. As a result, different task can only exchange data via files.

Thao et al. introduced the Swift [26] system to reduce the management issues which occur when a job involving numerous tasks has to be executed on a large, possibly

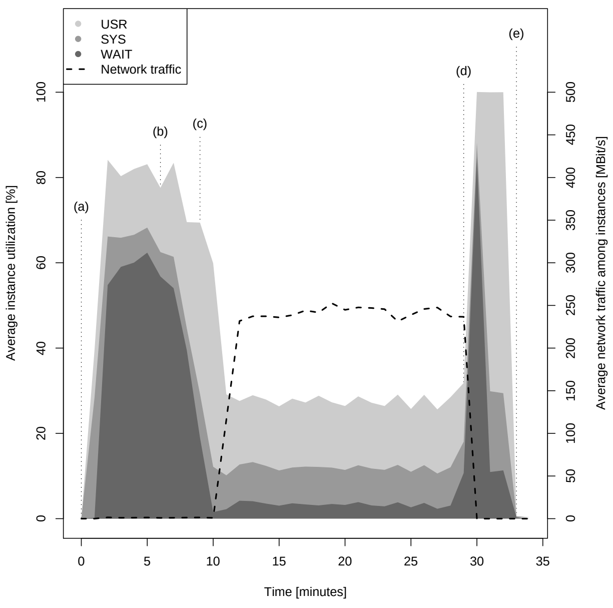

Figure 5: Results of the evaluation running Nephele

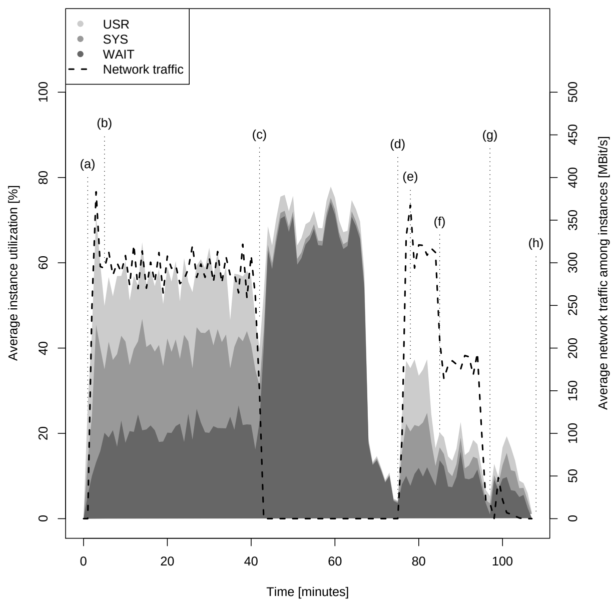

Figure 6: Results of the evaluation running Hadoop

unstructured, set of data. Building upon components like CoG Karajan [24], Falkon [20] and Globus [11], the authors present a scripting language which allows to create mappings between logical and physical data structures and to conveniently assign tasks to these.

The system our approach probably shares most similarities with is Dryad [13]. Dryad also runs jobs described as DAGs and offers to connect the involved tasks through either file, network or in-memory channels. However, it assumes an execution environment which consists of a fixed set of homogeneous worker nodes. Dryad scheduler is designed to distribute tasks across the available compute nodes in a way that optimizes the throughput of the overall cluster. It does not include the notion of processing cost for particular jobs.

6. CONCLUSION

In this paper we discussed the challenges and opportunities for efficient parallel data processing in cloud environments and presented Nephele, the first data processing framework to exploit the dynamic resource provisioning offered by today's compute clouds. We described Nephele's basic architecture and presented a performance comparison to the well-established data processing framework Hadoop. The performance evaluation gave a first impression on how the ability to assign specific virtual machine types to specific tasks of a processing job as well as the possibility to automatically allocate/deallocate virtual machines in the course of a job execution can help to improve the overall resource utilization and, consequently, reduce the processing cost.

With a framework like Nephele at hand, there are a variety of open research issue which we plan to address for future work. In particular, we are interested to find out how Nephele can learn from job executions to construct more efficient Execution Graphs (either in terms of cost or execution time) for upcoming jobs. In its current version, Nephele strongly depends on the user's annotations to create an efficient Execution Graph. For future versions we envision our framework to detect performance bottlenecks independently and to improve the affected parts of the Execution Graph for future runs. Constructing these kinds of feedback loops has brought up remarkable results in field of database systems [22], however, it is still unclear to what extent these approaches can be applied to cloud computing, too.

In general, we think our work represents an important contribution to the growing field of cloud computing services and points out exciting new directions in the field of parallel data processing.

7. REFERENCES

[1] Amazon Web Services LLC. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`, 2009.

[2] Amazon Web Services LLC. Amazon Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce/`, 2009.

[3] Amazon Web Services LLC. Amazon Simple Storage Service. `http://aws.amazon.com/s3/`, 2009.

[4] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Kakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification (WS-Agreement). Technical report, Open Grid Forum, 2007.

[5] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[6] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040, New York, NY, USA, 2007. ACM.

[7] M. Coates, R. Castro, R. Nowak, M. Gadhiok, R. King, and Y. Tsang. Maximum likelihood network

topology identification from edge-based unicast measurements. *SIGMETRICS Perform. Eval. Rev.*, 30(1):11–20, 2002.

[8] R. Davoli. VDE: Virtual Distributed Ethernet. *Testbeds and Research Infrastructures for the Development of Networks & Communities, International Conference on*, 0:213–220, 2005.

[9] J. Dean and S. 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.

[10] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, 2005.

[11] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl. Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[12] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. 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.

[14] A. Kivity. kvm: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[15] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. Eucalyptus: A Technical Report on an Elastic Utility Computing Architecture Linking Your Programs to Useful Systems. Technical report, University of California, Santa Barbara, 2008.

[16] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[17] O. O'Malley and A. C. Murthy. Winning a 60 Second Dash with a Yellow Elephant. Technical report, Yahoo!, 2009.

[18] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[19] I. Raicu, I. Foster, and Y. Zhao. Many-task computing for grids and supercomputers. In *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, pages 1–11, Nov. 2008.

[20] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a Fast and Light-weight tasK executiON framework. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[21] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[22] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. Leo - DB2's LEarning Optimizer. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 19–28, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[23] The Apache Software Foundation. Welcome to Hadoop! `http://hadoop.apache.org/`, 2009.

[24] G. von Laszewski, M. Hategan, and D. Kodeboyina. *Workflows for e-Science Scientific Workflows for Grids.* Springer, 2007.

[25] T. White. *Hadoop: The Definitive Guide.* O'Reilly Media, 2009.

[26] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, Reliable, Loosely Coupled Parallel Computation. In *Services, 2007 IEEE Congress on*, pages 199–206, July 2007.