Exploiting Dynamic Resource Allocation for Efficient Parallel Data Processing in the Cloud

Daniel Warneke and Odej Kao

Abstract—In recent years ad-hoc parallel data processing has emerged to be one of the killer applications for Infrastructure-as-a-Service (IaaS) clouds. 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 have been designed for static, homogeneous cluster setups and disregard the particular nature of a cloud. Consequently, 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 research project Nephele. Nephele is the first data processing framework to explicitly exploit the dynamic resource allocation offered by today's IaaS clouds for both, task scheduling and execution. Particular tasks of a processing job can be assigned to different types of virtual machines which are automatically instantiated and terminated during the job execution. Based on this new framework, we perform extended evaluations of MapReduce-inspired processing jobs on an IaaS cloud system and compare the results to the popular data processing framework Hadoop.

Index Terms—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. Classic 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 based on a large number of commodity servers. Problems like processing crawled documents 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. Examples are Google's MapReduce [9], Microsoft's Dryad [14], 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 [20]. Although these systems differ in design, their programming models 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.

• *The authors are with the Berlin University of Technology, Einsteinufer 17, 10587 Berlin, Germany.*
E-mail: daniel.warneke@tu-berlin.de, odej.kao@tu-berlin.de

For companies that only have to process large amounts of data occasionally 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 so-called Infrastructure-as-a-Service (IaaS) clouds, like Amazon EC2 [1], let their customers allocate, access, and control a set of virtual machines (VMs) which run inside their data centers and only charge them for the period of time the machines are allocated. The VMs 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 VM abstraction of IaaS clouds fits the architectural paradigm assumed by the data processing frameworks described above, projects like Hadoop [25], a popular open source implementation of Google's MapReduce framework, already have begun to promote using their frameworks in the cloud [29]. 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 VMs 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 pro-

cessing in 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.

This paper is an extended version of [27]. It includes further details on scheduling strategies and extended experimental results. The 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 in 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 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 IaaS clouds can certainly be used to create such cluster-like setups, much of their flexibility remains unused.

One of an IaaS cloud's key features is the provisioning of compute resources on demand. New VMs 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 VMs 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 workflow. E.g., a framework exploiting the possibilities of a cloud could start with a single VM which analyzes an incoming job and then advises the cloud to directly start the required VMs 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.

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 VMs 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 VM is no longer needed 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 VMs, since it is unclear if 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 VM 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 [24].

2.2 Challenges

The cloud's virtualized nature helps to enable promising new use cases for efficient parallel data processing. However, it also imposes new challenges compared to classic cluster setups. The major challenge we see is the cloud's *opaqueness* 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 [29]. 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 IaaS 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 in a cloud and use it for topology-aware scheduling, the obtained information would not necessarily remain valid for the entire processing time. VMs may be migrated for administrative purposes between different locations inside the 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 VM in the cloud. This may involve allocating fewer, but more powerful VMs 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 VM with eight cores instead of eight distinct single-core machines. However, currently no data processing framework 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 classic master-worker pattern as illustrated in Fig. 1.

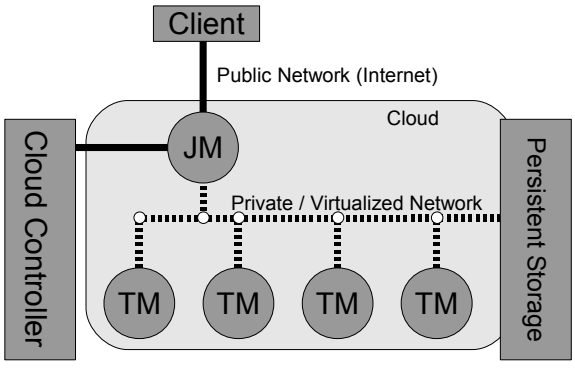

Fig. 1. Structural overview of Nephele running in an Infrastructure-as-a-Service (IaaS) cloud

Before submitting a Nephele compute job, a user must start a VM in 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 interface the cloud operator provides to control the instantiation of VMs. We call this interface the *Cloud Controller*. By means of the Cloud Controller the Job Manager can allocate or deallocate VMs according to the current job execution phase. We will comply with common Cloud computing terminology and refer to these VMs as *instances* for the remainder of this paper. The term *instance type* will be used to differentiate between VMs with different hardware characteristics. E.g., the instance type "m1.small" could denote VMs with one CPU core, one GB of RAM, and a 128 GB disk while the instance type "c1.xlarge" could refer to machines with 8 CPU cores, 18 GB RAM, and a 512 GB disk.

The actual execution of tasks which a Nephele job consists of is carried out by a set of instances. Each instance runs a so-called *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 job's particular tasks, how many and what type of instances the job should be executed on, and when the respective instances must be allocated/deallocated to ensure a continuous but cost-efficient processing. Our current strategies for these decisions are highlighted at the end of this section.

The newly allocated instances boot up with a previously compiled VM 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 VM 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 [14], 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 classic data

combining functions like, e.g., join operations [6]. Second and more importantly, though, the DAG's edges explicitly model the communication paths of the processing job. As long as the particular 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. Programmers can define arbitrary types of records. From a programmer's perspective records enter and leave the task program through input or output gates. Those input and output gates can be considered endpoints of the DAG's edges which are defined in the following step. Regular tasks (i.e. tasks which are later assigned to inner vertices of the DAG) must have at least one or more input and output gates. Contrary to that, tasks which either represent the source or the sink of the data flow must not have input or output gates, respectively.

After having specified the code for the particular tasks of the job, the user must define the DAG to connect these tasks. We call this DAG the *Job Graph*. The Job Graph maps each task to a vertex and determines the communication paths between them. The number of a vertex's incoming and outgoing edges must thereby comply with the number of input and output gates defined inside the tasks. In addition to the task to execute, input and output vertices (i.e. vertices with either no incoming or outgoing edge) can be associated with a URL pointing to external storage facilities to read or write input or output data, respectively. Figure 2 illustrates the simplest possible Job Graph. It only consists of one input, one task, and one output vertex.

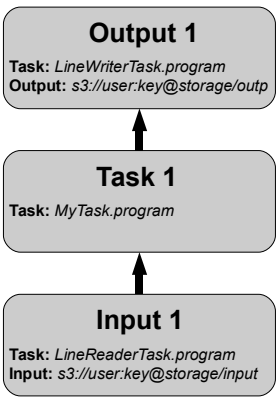

Fig. 2. An example of a Job Graph in Nephele

One major design goal of Job Graphs has been simplicity: Users should be able to describe tasks and their relationships on an abstract level. Therefore, the Job Graph does not explicitly model task parallelization and the mapping of tasks to instances. However, users who wish to influence these aspects can provide 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 subtask is assigned to a separate instance. In case several subtasks 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 channel type. Before executing a job, Nephele requires all edges of the original Job Graph to be replaced by at least one channel of a specific type. The channel type dictates how records are transported from one subtask to another at runtime. Currently, Nephele supports network, file, and in-memory channels. The choice of the channel type can have several implications on the entire job schedule. A more detailled disussion on this is provided in the next subsection.
- **Instance type:** A subtask 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 applies default strategies which are discussed later on in this section.

Once the Job Graph is specified, the user submits it to the Job Manager, together with the credentials he has obtained from his cloud operator. The credentials are required since the Job Manager must allocate/deallocate instances 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. It explicitly models task parallelization and the mapping of tasks to instances. 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). Task 1 is e.g. split into two parallel subtasks which are both connected to the task Output 1 via file channels and are all scheduled to run on the same instance. The exact structure of the Execution Graph is explained in the following:

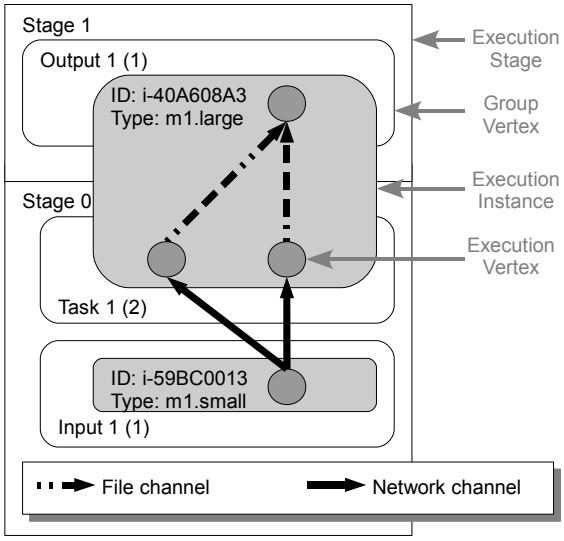

Fig. 3. An Execution Graph created from the original Job Graph

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 fine-grained 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 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 in an IaaS cloud, Nephele allows to allocate/deallocate instances in the course of the processing job, when some subtasks 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 the subtasks included in the preceding stages 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 VMs 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 VM. 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 processing 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 distributed among them 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 task parallelization, 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 subtasks. The actual number of channels that are connected to a subtask at runtime is hidden behind the task's respective input and output gates. However, the user code can determine the number if necessary.

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. An in-memory channel typically represents the fastest way to transport records in Nephele, however, it also implies most scheduling restrictions: The two connected subtasks must be scheduled to run on the same instance and run 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.

3.4 Parallelization and Scheduling Strategies

As mentioned before, 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. Discussing this subject in detail would go beyond the scope of this paper. However, we want to outline our basic approaches in this subsection:

Unless the user provides any job annotation which contains more specific instructions we currently pursue a simple default strategy: Each vertex of the Job Graph is transformed into one Execution Vertex. 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 one with the lowest price per time unit available in the IaaS cloud.

One fundamental idea to refine the scheduling strategy for recurring jobs is to use feedback data. We developed a profiling subsystem for Nephele which can continously monitor running tasks and the underlying instances. Based on the Java Management Extensions (JMX) the profiling subsystem is, among other things, capable of breaking down what percentage of its processing time a task thread actually spends processing user code and what percentage of time it has to wait for data. With the collected data Nephele is able to detect both computational as well as I/O bottlenecks. While computational bottlenecks suggest a higher degree of parallelization for the affected tasks, I/O bottlenecks provide hints to switch to faster channel types (like in-memory channels) and reconsider the instance assignment. Since Nephele calculates a cryptographic signature for each task, recurring tasks can be identified and the previously recorded feedback data can be exploited.

At the moment we only use the profiling data to detect these bottlenecks and help the user to choose reasonable annotations for his job. Figure 4 illustrates the graphical job viewer we have devised for that purpose. It provides immediate visual feedback about the current utilization of all tasks and cloud instances involved in the computation. A user can utilize this visual feedback to improve his job annotations for upcoming job executions. In more advanced versions of Nephele we envision the system to automatically adapt to detected bottlenecks, either between consecutive executions of the same job or even during job execution at runtime.

While the allocation time of cloud instances is determined by the start times of the assigned subtasks, there are different possible strategies for instance deallocation. In order to reflect the fact that most cloud providers charge their customers for instance usage 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.

Besides the use of feedback data we recently complemented our efforts to provide reasonable job annotations automatically by a higher-level programming model layered on top of Nephele. Rather than describing

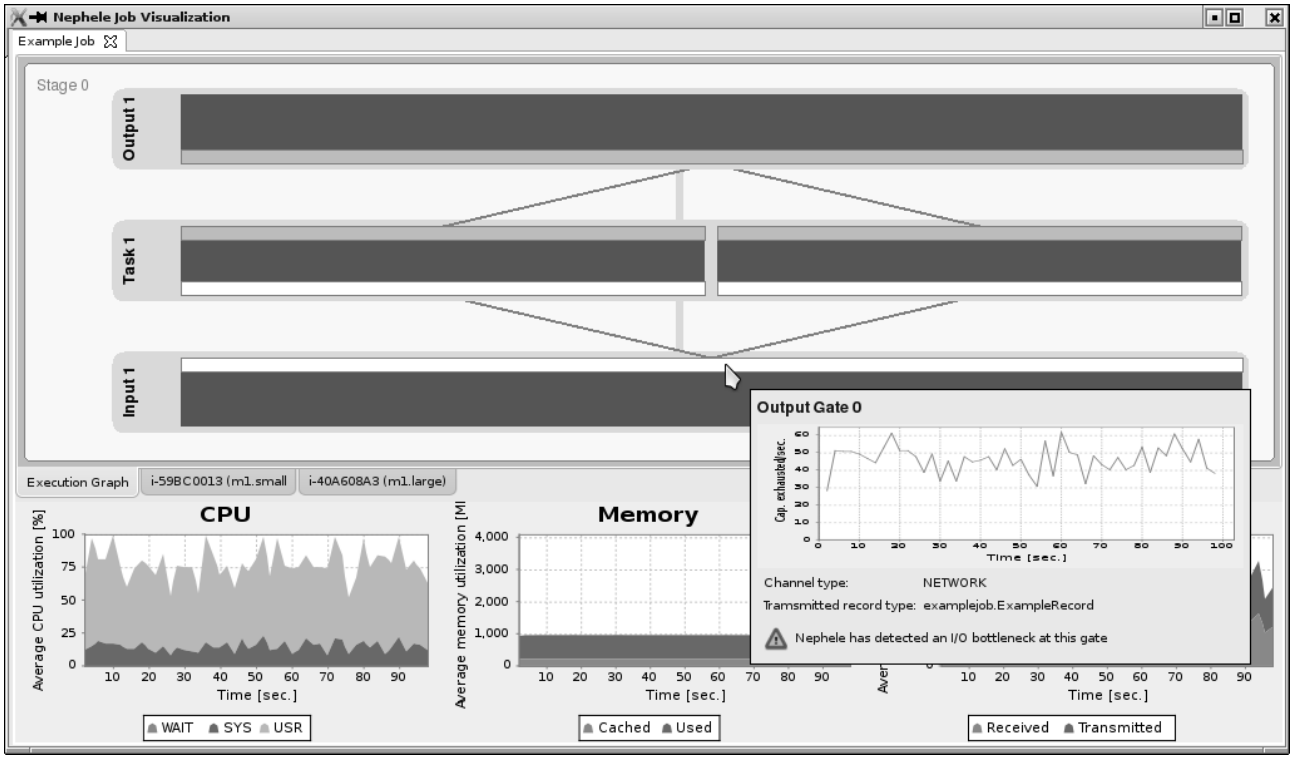

Fig. 4. Nephele's job viewer provides graphical feedback on instance utilization and detected bottlenecks

jobs as arbitrary DAGs, this higher-level programming model called PACTs [4] is centered around the concatenation of second-order functions, e.g. like the map and reduce function from the well-known MapReduce programming model. Developers can write custom first-order functions and attach them to the desired second-order functions. The PACTs programming model is semantically richer than Nephele's own programming abstraction. E.g. it considers aspects like the input/output cardinalities of the first order functions which is helpful to deduce reasonable degrees of parallelization. More details can be found in [4].

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 an 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 fewer instances in current IaaS 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 for both frameworks 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 these k smallest numbers. The job is a classic representative for a variety of data analysis jobs whose particular tasks vary in their complexity and hardware demands. 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.

We implemented the described sort/aggregate task for three different experiments. For the first experiment, we implemented the task as a sequence of MapReduce programs and executed it using Hadoop on a fixed set of instances. For the second experiment, we reused the same MapReduce programs as in the first experiment but devised a special MapReduce wrapper to make these programs run on top of Nephele. The goal of this experiment was to illustrate the benefits of dynamic resource allocation/deallocation while still maintaining the MapReduce processing pattern. Finally, as the third experiment, we discarded the MapReduce pattern and implemented the task based on a DAG to also highlight the advantages of using heterogeneous instances.

For all three experiments, 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. The cut-off variable 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

All three experiments were conducted on our local IaaS cloud of commodity servers. Each server is equipped with two Intel Xeon 2.66 GHz CPUs (8 CPU cores) and a total main memory of 32 GB. All servers are connected through regular 1 GBit/s Ethernet links. The host operating system was Gentoo Linux (kernel version 2.6.30) with KVM [15] (version 88-r1) using virtio [23] to provide virtual I/O access.

To manage the cloud and provision VMs on request of Nephele, we set up Eucalyptus [16]. Similar to Amazon EC2, Eucalyptus offers a predefined set of instance types a user can choose from. During our experiments we used two different instance types: The first instance type was "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 about 0.10 $, or 0.80 $ per hour (September 2009), respectively.

The images used to boot up the instances contained Ubuntu Linux (kernel version 2.6.28) with no additional software but a Java runtime environment (version 1.6.0_13), 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 [18]. In order to make the data accessible to Hadoop, we started an HDFS [25] 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 all three experiments (Hadoop and Nephele), we have chosen to ignore it in the following performance discussion.

4.2 Experiment 1: MapReduce and Hadoop

In order to execute the described sort/aggregate task with Hadoop we created three different MapReduce programs which were 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 [18]. The result of this first MapReduce job was a set of files containing sorted integer numbers. Concatenating these files yielded the fully sorted sequence of 10^9 numbers.

The second and third MapReduce jobs operated on the sorted data set and performed the data aggregation. Thereby, the second MapReduce job selected the first output files from the preceding sort job which, just by their file size, had to contain the smallest $2 \cdot 10^8$ numbers of the initial data set. The map function was fed with the selected files and emitted 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 ensured that the emitted numbers were distributed evenly among the n reducers in the system. Each reducer then calculated the average of the received $\frac{2 \cdot 10^8}{n}$ integer numbers. The third MapReduce job finally read the n intermediate average values and aggregated 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. All of these instances were assigned to Hadoop throughout the entire duration of the experiment.

We configured Hadoop to perform best for the first, computationally most expensive, MapReduce job: In accordance to [18] 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 Experiment 2: MapReduce and Nephele

For the second experiment we reused the three MapReduce programs we had written for the previously described Hadoop experiment and executed them on top of Nephele. In order to do so, we had to develop a set of wrapper classes providing limited interface compatibility with Hadoop and sort/merge functionality. These wrapper classes allowed us to run the unmodified Hadoop MapReduce programs with Nephele. As a result, the data flow was controlled by the executed MapReduce programs while Nephele was able to govern the instance allocation/deallocation and the assignment of tasks to instances during the experiment. We devised this experiment to highlight the effects of the dynamic resource allocation/deallocation while still maintaining comparability to Hadoop as well as possible.

Figure 5 illustrates the Execution Graph we instructed Nephele to create so that the communication paths match the MapReduce processing pattern. For brevity, we omit a discussion on the original Job Graph. Following our experiences with the Hadoop experiment, we pursued the overall idea to also start with a homogeneous set of six "c1.xlarge" instances, but to reduce the number of allocated instances in the course of the experiment according to the previously observed workload. For this reason, the sort operation should be carried out on all six instances, while the first and second aggregation operation should only be assigned to two instances and to one instance, respectively.

The experiment's Execution Graph consisted of three Execution Stages. Each stage contained the tasks required by the corresponding MapReduce program. As stages can only be crossed via file channels, all interme-

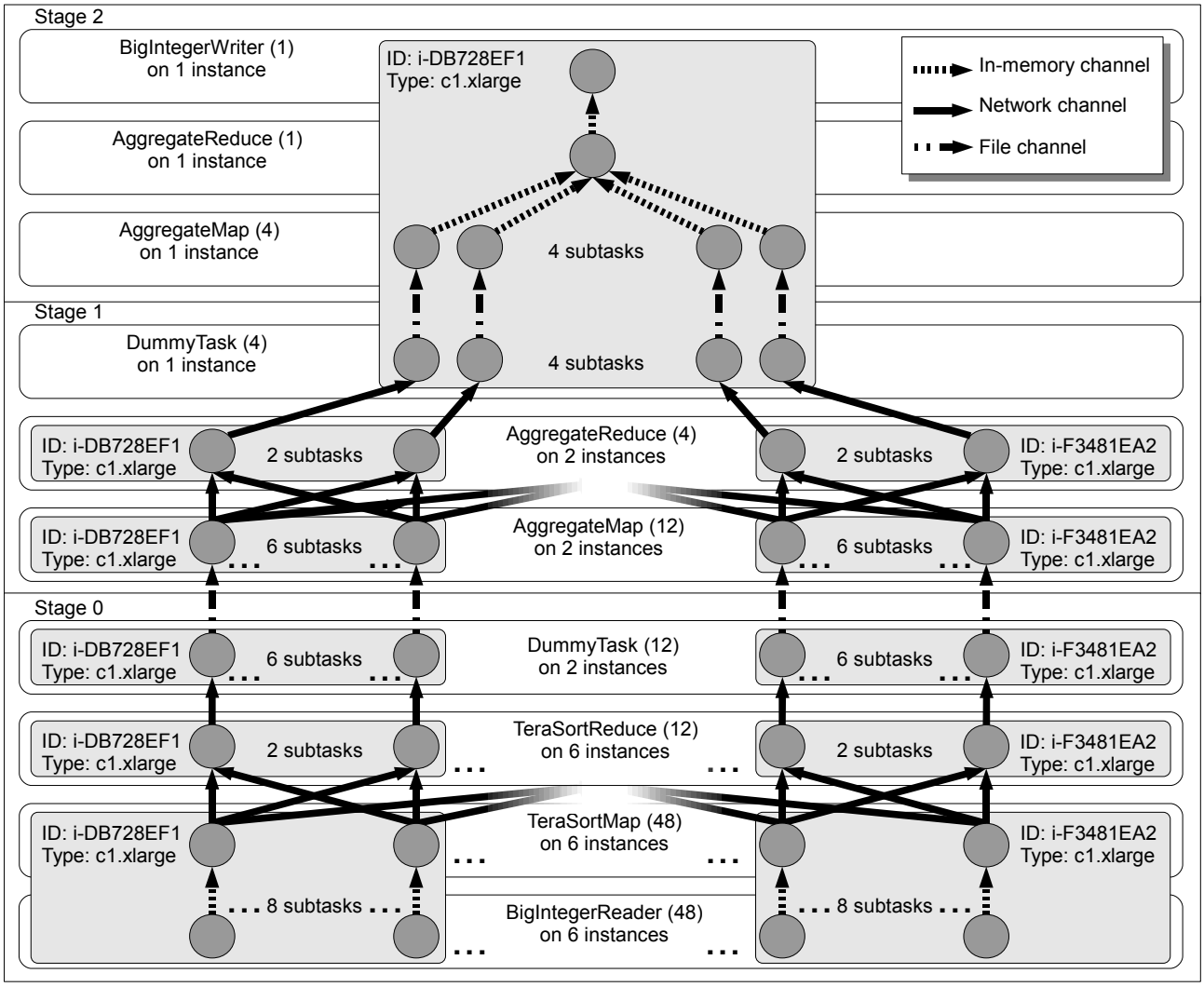

Fig. 5. The Execution Graph for Experiment 2 (MapReduce and Nephele)

diate data occurring between two succeeding MapReduce jobs was completely written to disk, like in the previous Hadoop experiment.

The first stage (Stage 0) included four different tasks, split into several different groups of subtasks and assigned to different instances. The first task, *BigIntegerReader*, processed the assigned input files and emitted each integer number as a separate record. The tasks *TeraSortMap* and *TeraSortReduce* encapsulated the TeraSort MapReduce code which had been executed by Hadoop's mapper and reducer threads before. In order to meet the setup of the previous experiment, we split the TeraSortMap and TeraSortReduce tasks into 48 and 12 subtasks, respectively, and assigned them to six instances of type "c1.xlarge". Furthermore, we instructed Nephele to construct network channels between each pair of TeraSortMap and TeraSortReduce subtasks. For the sake of legibility, only few of the resulting channels are depicted in Figure 5.

Although Nephele only maintains at most one physical TCP connection between two cloud instances, we devised the following optimization: If two subtasks that are connected by a network channel are scheduled to run on the same instance, their network channel is dynamically converted into an in-memory channel. That way we were able to avoid unnecessary data serialization and the resulting processing overhead. For the given MapReduce communication pattern this optimization accounts for approximately 20% less network channels.

The records emitted by the BigIntegerReader subtasks were received by the TeraSortMap subtasks. The TeraSort partitioning function, located in the TeraSortMap task, then determined which TeraSortReduce subtask was responsible for the received record, depending on its value. Before being sent to the respective reducer, the records were collected in buffers of approximately 44 MB size and were presorted in memory. Considering that each TeraSortMap subtask was connected to 12 TeraSortReduce subtasks, this added up to a buffer size of 574 MB, similar to the size of the in-memory file system we had used for the Hadoop experiment previously.

Each TeraSortReducer subtask had an in-memory

buffer of about 512 MB size, too. The buffer was used to mitigate hard drive access when storing the incoming sets of presorted records from the mappers. Like Hadoop, we started merging the first received presorted record sets using separate background threads while the data transfer from the mapper tasks was still in progress. To improve performance, the final merge step, resulting in one fully sorted set of records, was directly streamed to the next task in the processing chain.

The task *DummyTask*, the forth task in the first Execution Stage, simply emitted every record it received. It was used to direct the output of a preceding task to a particular subset of allocated instances. Following the overall idea of this experiment, we used the DummyTask task in the first stage to transmit the sorted output of the 12 TeraSortReduce subtasks to the two instances Nephele would continue to work with in the second Execution Stage. For the DummyTask subtasks in the first stage (Stage 0) we shuffled the assignment of subtasks to instances in a way that both remaining instances received a fairly even fraction of the $2 \cdot 10^8$ smallest numbers. Without the shuffle, the $2 \cdot 10^8$ would all be stored on only one of the remaining instances with high probability.

In the second and third stage (Stage 1 and 2 in Figure 5) we ran the two aggregation steps corresponding to the second and third MapReduce program in the previous Hadoop experiment. *AggregateMap* and *AggregateReduce* encapsulated the respective Hadoop code.

The first aggregation step was distributed across 12 AggregateMap and four AggregateReduce subtasks, assigned to the two remaining "c1.xlarge" instances. To determine how many records each AggregateMap subtask had to process, so that in total only the $2 \cdot 10^8$ numbers would be emitted to the reducers, we had to develop a small utility program. This utility program consisted of two components. The first component ran in the DummyTask subtasks of the preceding Stage 0. It wrote the number of records each DummyTask subtask had eventually emitted to a network file system share which was accessible to every instance. The second component, integrated in the AggregateMap subtasks, read those numbers and calculated what fraction of the sorted data was assigned to the respective mapper. In the previous Hadoop experiment this auxiliary program was unnecessary since Hadoop wrote the output of each MapReduce job back to HDFS anyway.

After the first aggregation step, we again used the DummyTask task to transmit the intermediate results to the last instance which executed the final aggregation in the third stage. The final aggregation was carried out by four AggregateMap subtasks and one AggregateReduce subtask. Eventually, we used one subtask of *BigIntegerWriter* to write the final result record back to HDFS.

4.4 Experiment 3: DAG and Nephele

In this third experiment we were no longer bound to the MapReduce processing pattern. Instead, we implemented the sort/aggregation problem as a DAG and tried to exploit Nephele's ability to manage heterogeneous compute resources.

Figure 6 illustrates the Execution Graph we instructed Nephele to create for this experiment. For brevity, we again leave out a discussion on the original Job Graph. Similar to the previous experiment, we pursued the 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 contained five distinct tasks, again split into different groups of subtasks. However, in contrast to the previous experiment, this one also involved instances of different types.

In order to feed the initial data from HDFS into Nephele, we reused the *BigIntegerReader* task. The records emitted by the BigIntegerReader subtasks were received by the second task, *BigIntegerSorter*, which attempted to buffer all incoming records into main memory. Once it had received all designated records, it performed an in-memory quick sort and subsequently continued 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*, received records from multiple input channels. Once it has read a record from all available input channels, it sorts the records locally and always emits the smallest number. The BigIntegerMerger tasks occured 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 occured in the Execution Graph, it was split into two subtasks. These two subtasks were assigned to two of the previously used "c1.xlarge" instances. The third occurrence of the task was assigned to new instance of the type "m1.small".

Since we abandoned the MapReduce processing pattern, we were able to better exploit Nephele's streaming pipelining characteristics in this experiment. Consequently, each of the merge subtasks was configured to stop execution after having emitted $2 \cdot 10^8$ records. The stop command was propagated to all preceding subtasks of the processing chain, which allowed the Execution Stage to be interrupted as soon as the final merge subtask had emitted the $2 \cdot 10^8$ smallest records.

The fourth task, *BigIntegerAggregater*, read the incoming records from its input channels and summed them up. It was also assigned to the single "m1.small" instance. Since we no longer required the six "c1.xlarge" instances to run once the final merge subtask had determined the $2 \cdot 10^8$ smallest numbers, we changed the communication channel between the final BigInte-

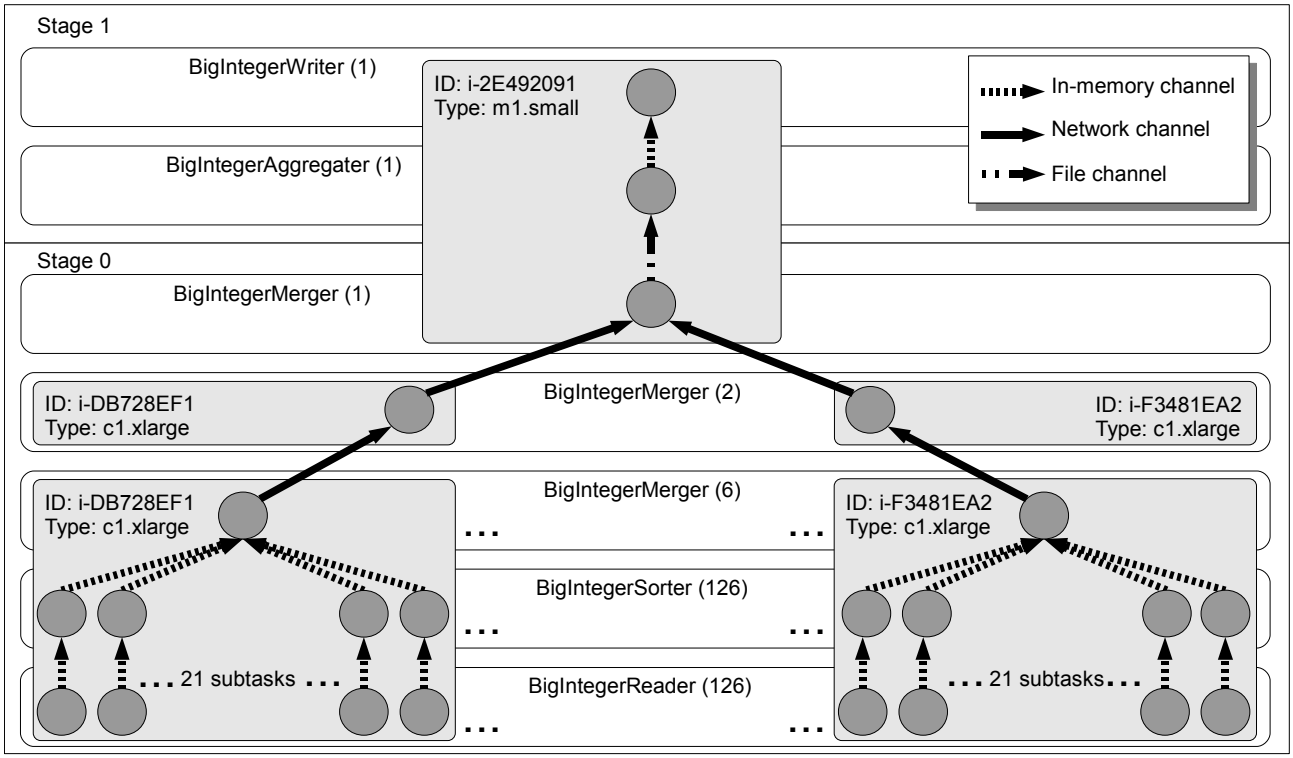

Fig. 6. The Execution Graph for Experiment 3 (DAG and Nephele)

gerMerger and BigIntegerAggregater subtask to a file channel. That way Nephele pushed the aggregation into the next Execution Stage and was able to deallocate the expensive instances.

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

4.5 Results

Figure 7, Figure 8 and Figure 9 show the performance results of our three experiment, respectively. All three plots illustrate the average instance utilization over time, i.e. the average utilization of all CPU cores in all 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 (WAIT). In order to illustrate the impact of network communication, the plots additionally show the average amount of IP traffic flowing between the instances over time.

We begin with discussing Experiment 1 (MapReduce and Hadoop): For the first MapReduce job, TeraSort, Figure 7 shows a fair resource utilization. During the map (point (a) to (c)) and reduce phase (point (b) to (d)) the overall system utilization ranges from 60 to 80%. This is reasonable since we configured Hadoop's MapReduce engine to perform best for this kind of task. 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 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. 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, all six expensive instances must be allocated throughout the entire experiment and unnecessarily contribute to the processing cost.

Figure 8 shows the system utilization for executing the same MapReduce programs on top of Nephele. For the first Execution Stage, corresponding to the TeraSort map and reduce tasks, the overall resource utilization is comparable to the one of the Hadoop experiment. During the map phase (point (a) to (c)) and the reduce phase (point (b) to (d)) all six "c1.xlarge" instances show an average utilization of about 80 %. However, after approximately 42 minutes, Nephele starts transmitting the sorted output stream of each of the 12 TeraSortReduce subtasks to the two instances which are scheduled to remain allocated for the upcoming Execution Stages. At the end of Stage 0 (point (d)), Nephele is aware that four of the six "c1.xlarge" are no longer required for the upcoming computations and deallocates them.

Since the four deallocated instances do no longer

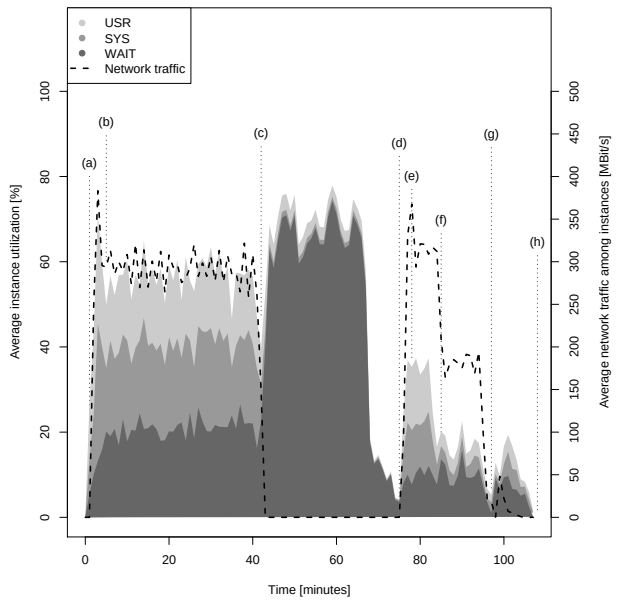

Fig. 7. Results of Experiment 1: MapReduce and Hadoop

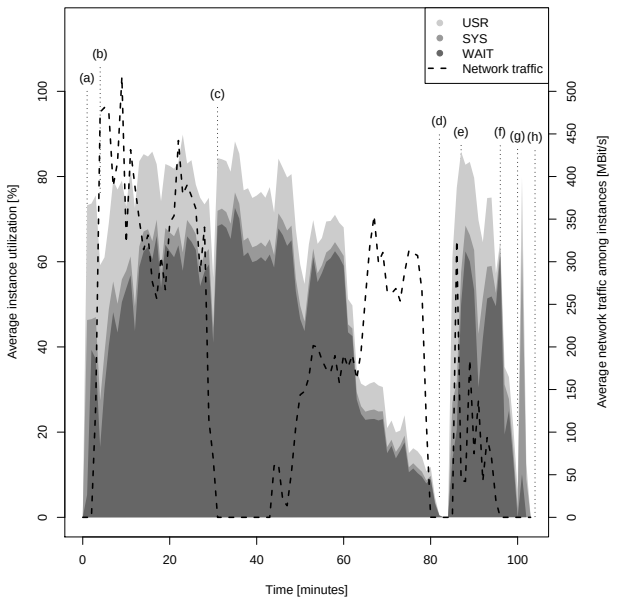

Fig. 8. Results of Experiment 2: MapReduce and Nephele

contribute to the number of available CPU cores in the second stage, the remaining instances again match the computational demands of the first aggregation step. During the execution of the 12 AggregateMap subtasks (point (d) to (f)) and the four AggregateReduce subtasks (point (e) to (g)), the utilization of the allocated instances is about 80%. The same applies to the final aggregation in the third Execution Stage(point (g) to (h)) which is only executed on one allocated "c1.xlarge" instance.

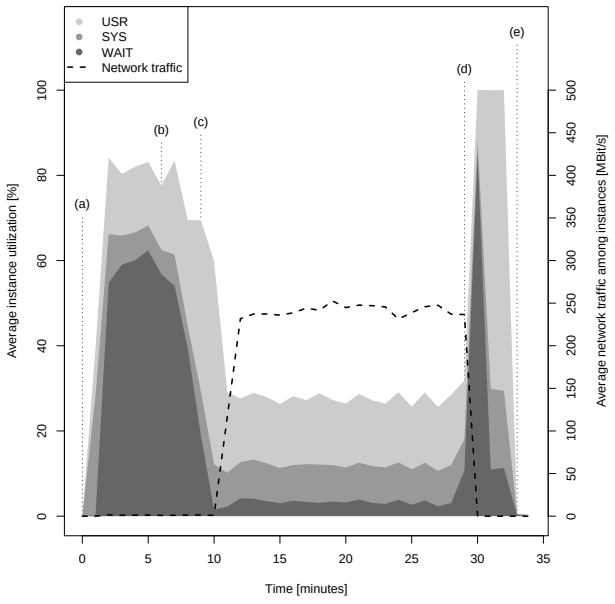

Fig. 9. Results of Experiment 3: DAG and Nephele

Finally, we want to discuss the results of the third experiment (DAG and Nephele) as depicted in Figure 9: 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" instances. After that period the computational power is no longer needed for the merge phase. 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 transferred to the inexpensive "m1.small" instance first. Here we identified the network to be the bottleneck, so much computational power remains unused during that transfer phase (from (c) to (d)). In general, this transfer penalty must be carefully considered when switching between different instance types during 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 their average. Again, 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.

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 MTC has been developed. Although these systems typically share common goals (e.g. to hide issues of parallelism or fault tolerance), they aim at different fields of application.

MapReduce [9] (or the open source version Hadoop [25]) 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 has 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 [19], [17].

MapReduce has been clearly designed for large static clusters. Although it can deal with sporadic node failures, the available compute resources 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 of allocated instances in 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 [13] as its execution engine. As a result, different task can only exchange data via files.

Thao et al. introduced the Swift [30] system to reduce the management issues which occur when a job involving numerous tasks has to be executed on a large, possibly unstructured, set of data. Building upon components like CoG Karajan [26], Falkon [21], and Globus [12], 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 [14]. Dryad also runs DAG-based jobs 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. The 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.

In terms of on-demand resource provising several projects arose recently: Dörnemann et al. [11] presented an approach to handle peak-load situations in BPEL workflows using Amazon EC2. Ramakrishnan et al. [22] discussed how to provide a uniform resource abstraction over grid and cloud resources for scientific workflows. Both projects rather aim at batch-driven workflows than the data-intensive, pipelined workflows Nephele focuses on. The FOS project [28] recently presented an operating system for multicore and clouds which is also capable of on-demand VM allocation.

6 Conclusion

In this paper we have 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 IaaS clouds. We have described Nephele's basic architecture and presented a performance comparison to the well-established data processing framework Hadoop. The performance evaluation gives 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 issues, which we plan to address for future work. In particular, we are interested in improving Nephele's ability to adapt to resource overload or underutilization during the job execution automatically. Our current profiling approach builds a valuable basis for this, however, at the moment the system still requires a reasonable amount of user annotations.

In general, we think our work represents an important contribution to the growing field of Cloud computing

services and points out exciting new opportunities in the field of parallel data processing.

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] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *SoCC '10: Proceedings of the ACM Symposium on Cloud Computing 2010*, pages 119–130, New York, NY, USA, 2010. ACM.

[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] T. Dornemann, E. Juhnke, and B. Freisleben. On-Demand Resource Provisioning for BPEL Workflows Using Amazon's Elastic Compute Cloud. In *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 140–147, Washington, DC, USA, 2009. IEEE Computer Society.

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

[13] 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.

[14] 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.

[15] A. Kivity. kvm: the Linux Virtual Machine Monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[16] 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.

[17] 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.

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

[19] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[20] 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.

[21] 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.

[22] L. Ramakrishnan, C. Koelbel, Y.-S. Kee, R. Wolski, D. Nurmi, D. Gannon, G. Obertelli, A. YarKhan, A. Mandal, T. M. Huang, K. Thyagaraja, and D. Zagorodnov. VGrADS: Enabling e-Science Workflows on Grids and Clouds with Fault Tolerance. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[23] R. Russell. virtio: Towards a De-Facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[24] 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.

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

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

[27] D. Warneke and O. Kao. Nephele: Efficient Parallel Data Processing in the Cloud. In *MTAGS '09: Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, pages 1–10, New York, NY, USA, 2009. ACM.

[28] D. Wentzlaff, C. G. III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal. An Operating System for Multicore and Clouds: Mechanisms and Implementation. In *SoCC '10: Proceedings of the ACM Symposium on Cloud Computing 2010*, pages 3–14, New York, NY, USA, 2010. ACM.

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

[30] 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.

Daniel Warneke is a research assistant at the Berlin University of Technology, Germany. He received his Diploma and BS degrees in computer science from the University of Paderborn, in 2008 and 2006, respectively. Daniel's research interests center around massively-parallel, fault-tolerant data processing frameworks on Infrastructure-as-a-Service platforms. Currently, he is working in the DFG-funded research project Stratosphere.

Odej Kao is full professor at the Berlin University of Technology and director of the IT center tubIT. He received his PhD and his habilitation from the Clausthal University of Technology. Thereafter, he moved to the University of Paderborn as an associated professor for operating and distributed systems. His research areas include Grid Computing, service level agreements and operation of complex IT systems. Odej is a member of many program committees and has published more than 190 papers.