Maestro: a Self-Organizing Peer-to-Peer Dataflow Framework Using Reinforcement Learning

Kees van Reeuwijk
Vrije Universiteit Amsterdam
de Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
reeuwijk@few.vu.nl

ABSTRACT

In this paper we describe Maestro, a dataflow computation framework for Ibis, our Java-based grid middleware. The novelty of Maestro is that it is a *self-organizing peer-to-peer* system, meaning that it distributes the tasks in a flow over the available nodes based on local decisions on each node, without any central coordination. As a result, the computations are more scalable, more resilient against failing nodes, and less sensitive to communication latencies.

Maestro uses a task distribution approach based on *reinforcement learning*, a learning mechanism where the positive outcome of a choice makes it more likely that the same choice repeated in the future. Maestro selects the most efficient node for each stage in the computation based on the observed computation and communication times. To ensure agility, the selection decisions are made as late as possible without letting the nodes fall idle. Using this task distribution algorithm, the nodes can be used efficiently, even in a heterogeneous system with failure-prone nodes communicating through high-latency connections.

Categories and Subject Descriptors

C.1.4 [**Parallel Architectures**]: distributed architectures; D.3.2 [**Language Classifications**]: Data-flow languages; I.2.6 [**Learning**]: Parameter learning

General Terms

Algorithms,Experimentation,Management,Performance

Keywords

Peer to peer,reinforcement learning,self organizing

1. INTRODUCTION

There is an urgent need for a reliable parallel programming framework that provides efficient and robust computation even on large systems with long and unpredictable communication latencies and heterogeneous nodes. Getting efficient parallel execution on carefully designed homogeneous clusters at a single site is now a fairly well understood problem, but even there efficient use of large numbers of nodes can be problematic: once a computation involves hundreds or thousands of nodes, unexpected bottlenecks and inefficiencies become important enough to limit performance. Thus, *scalability* is a problem, even on dedicated homogeneous clusters.

Efficient parallel execution is even more problematic when clusters at multiple sites are used. Long-distance links have unavoidable latencies, may have a smaller capacity, and are often unreliable due to other communication load. Moreover, firewalls or NAT functionality often impose restrictions on the communication, meaning that between some nodes direct communication is simply not possible. Finally, the clusters at the different sites will almost certainly have nodes with different specifications, meaning that performance differences between the nodes must be taken into account.

Another issue is that with increasing numbers of nodes the probability of a node failing increases: a node can malfunction, be burdened with other computations, or the reservation of the node simply expires. As long as the failure rate is low, a computation can be restarted if necessary, but with higher failure rates this becomes impractical, and we need software that is able to cope with these changes. Similarly, it is helpful if the software allows new nodes to be added during the computation, a feature we call *maleability*.

The ultimate in such heterogeneous, distributed, and unreliable systems are volunteer computing grids such as BOINC [2]. Here any productivity is impossible if the heterogeneity, scalability, fallibility, and maleability of the system are not taken into account. In such systems it is also necessary to take *malicious* nodes into account, that participate in the computation, but give incorrect results. In this paper we ignore the problems of dealing with malicious nodes.

Efficient use of heterogeneous processors is also becoming an important issue for mainstream single-computer systems: GPU-assisted systems are also heterogeneous parallel systems. Systems on a Chip such as the Cell processor have multiple heterogeneous processors on a single chip today. Perhaps most importantly, with increased core counts in mainstream architectures it will become more and more tempting to add specialized cores for important high-performance tasks such as network or video processing, creating another heterogeneous processor system. Such systems are only viable if a robust method is available to use all processors efficiently. Although latency and fault tolerance are currently less significant issues on this level, with increased core counts and increasing integration their significance will only increase.

The goal of our long-running Ibis project [21, 18] is to provide a framework that addresses all these problems. Ibis offers robust and scalable program deployment, file access, message passing,

HPDC'09, June 11–13, 2009, Munich, Germany.

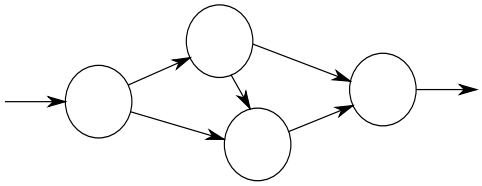

Figure 1: A flow graph for dataflow computation. The circles represent computing nodes, the arrows represent the flow of data between these nodes.

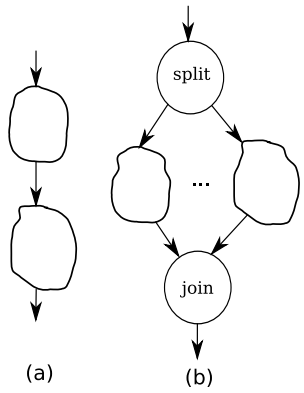

Figure 2: The composition rules of flow graphs allowed by Maestro: (a) sequential, and (b) split/join.

and other essential services for distributed computing. It also provides support for a number of high-level programming models, for example for divide-and-conquer problems [16].

In this paper we introduce a new programming model for Ibis: *Maestro*, a framework for *dataflow computation*, also known as *stream computation*. In this model computations are described as the flow of data between computing nodes, see Fig. 1. In general the graph of such a computation can be of arbitrary shape, but we restrict ourselves to SP graphs, where the graphs are constructed as either a sequence of two flow graphs, or a split/join operation on flow graphs (see Fig. 2). In a split/join graph the input is distributed over multiple sub-graphs, and the outputs of these graphs are combined ('joined') into the output of the split/join graph. Although these structural restrictions on the flow graph may introduce some inefficiencies, the effect is generally limited [11]. In this paper we concentrate on the self-configuring properties of the system, and restrict ourselves further to linear sequences of operations. We call each such a sequence a *job*, and each operation in a job a *task*.

Any central component in a system forms a bottleneck that potentially degrades the performance of the entire system, and a failure point that degrades the fault tolerance of the computation. One effective way eliminate such bottlenecks is to use a grid-based programming framework that does not require central coordination, but is entirely peer-to-peer. In this paper we will show that task scheduling for dataflow computations is possible in such a decentralized fashion. Every node in the system tries to select the best node for a task, based on the estimated time to completion of the remaining tasks on the available nodes. The necessary performance information is collected by the nodes themselves, exchanged between nodes through gossiping, and communicated between nodes along with other communication. Although Maestro supports the use of benchmarks to obtain initial estimates, during execution the performance information is based on real transmission of data and real execution of tasks. Once the system learns that a particular node is more efficient for a certain task, it will tend to use that node for subsequent task instances. However, due to uncertainty in the performance measurements, changing circumstances, and memory use on a node, the commitment of work to the nodes is carefully limited.

We will show that with this new, and remarkably simple, task distribution approach nodes can be used efficiently, even if they are heterogeneous, failure-prone, and linked with high-latency connections. This allows us to support dataflow computation on large heterogeneous systems, enabling a large range of important applications in areas such as video processing, signal processing, monitoring, etc.

In the remainder of this paper, we will first describe the context and inspiration of this work. We will then describe the used learning algorithm in detail, and describe a number of experiments to evaluate the performance of the system.

2. CONTEXT

Reinforcement learning is a learning technique where a successful outcome of a choice makes it more likely that the same choice is made in the future. This technique is frequently used in nature. A prominent example is that some species of ants mark the trail from an interesting food source with pheromones; foraging ants are more likely to follow a trail with a strong pheromone marking.

One algorithm based on reinforcement learning is the *Q-learning* algorithm [22]. It assumes a finite-state machine where with each state transition there is an associated reward. The goal of the algorithm is to find a series of state transitions to a given end state, where the sum of the transition rewards is maximal. The system learns by recording at each state for each possible transition the cumulative reward of that choice. This learning can be done in a separate session where at each state random choices are made, but it is also possible to learn during operation by allowing the system to sometimes make non-optimal choices.

The Q-learning algorithm has been used for routing in telecommunication networks [23]. In particular, Boyan et al. describe the Q-routing algorithm [4], based on the Q-learning algorithm. In their experiments, Q-routing performed similar to Shortest-Path routing under low network loads, and significantly better under higher network loads. For Maestro we use a conceptually similar algorithm, but the different applications require different concrete algorithms.

Another popular approach to peer-to-peer resource scheduling is to use an economic model [6] where workers 'bid' for work from work sources, and work sources select the most economic workers for their tasks. Thus, a supply-and-demand mechanism is used to find efficient schedules. Although the Maestro algorithm could largely also be formulated in economic terms, and although it is tempting to formulate a problem in terms that are intuitively understood by almost everyone, it is not clear that this would bring any advantages. We are not aware of any significant advantages of the economic models that are used in this context over Maestro's algorithm. Moreover, for the more complicated economic models it is not always clear that efficient solutions in economic terms are also efficient in terms of the real goal of the system. Finally, such systems often require resource brokers that mediate between supply and demand. Even if there is more than just a single central broker, such brokers are special nodes that are potential bottlenecks and failure points that degrade the advantages of peer-to-peer systems.

Chakravarti et al. [3] describe an 'organic grid' that is in its

goals very similar to Maestro. Their implementation uses a tree-shaped overlay network to organize the nodes, whereas in Maestro the communication patterns between the nodes are an emergent property of the system. Moreover, their system is restricted to a computation with a single type of independent task, although they have described some application-specific generalizations [9]. In contrast, Maestro targets high-throughput dataflow (stream) processing applications.

Grid middleware systems such as Condor [20] can also base their scheduling decisions on performance information about the individual nodes. However, such information is typically only specifies basic system and performance characteristics such as memory size, number of processors in the system, and a crude indication of performance based on a general benchmark. Even if a specific benchmark result is available that is relevant to a particular computation, such an estimate does not reflect more dynamic changes in the performance such as fluctuations in processor or network load. Moreover, such scheduling decisions are made centrally.

Dataflow systems are certainly not new [24]. IBM's System S [12] is a research system for stream processing that is now being commercialized. Other such systems are StreamFlex [19], Dataflow Java [14], Borealis [1], and Symphony [15]. Our contribution is to provide a self-organizing implementation of dataflow processing using reinforcement learning.

The dataflow approach is a natural match for visual programming environments, where the user defines a computation by drawing boxes and connections. A popular example is *LabView*, or more precisely *G*, the visual language of LabView. The area of *end-user programming*, programming languages designed for use by 'naive' users, is also dominated by (visual) dataflow languages [13].

Workflow systems such as Triana [8], GridBus [5], Gridflow [7], and Taverna [17] are very similar to dataflow systems, but are usually designed for course-grained computations, and are implemented as web services. Scheduling is usually done centrally, based on global knowledge, whereas in Maestro the scheduling is done locally, and can be based on imperfect global knowledge.

3. MAESTRO'S SCHEDULING ALGORITHM

In Maestro, a computation is executed by a set of nodes that may grow or shrink during the computation. A single node may use more than one processor (core) for the computation. The software that is run on each node consists of three components:

- A **master**, which distributes tasks over the available workers.
- A **worker**, which executes the tasks it receives from the masters in the system.
- A **gossiper**, which distributes performance information.

A job is started by placing a task instance for the first task in the job in the master queue of one of the nodes. Part of this task instance is the input for the task. Once the task instance is in the master queue, the following happens:

- The master regularly checks whether there is a worker on one of the nodes ready to handle a task of the particular type. If so, and if the task instance is the first of its type in the queue, the master selects the most efficient node for the task, and it sends the task instance to that node.
- The worker at the selected node receives the task, and places it in its queue.
- Once the task instance reaches the front of the worker queue, and once there is a ready processor, the task instance is executed, using the input that is part of the task instance.
- Once the execution has completed, the output of the execution is dispatched. If the executed task was the last task for that job, the output is sent to the node that submitted the first task of the job; the output becomes the output of the entire job. If there is a next task to be executed, the output becomes the input for a new task instance that is placed in the local master queue, and the entire process repeats for the new task instance.

Both the master and the worker maintain a queue that gives priority to task instances from earlier jobs. By prioritizing earlier jobs we ensure that once a job has entered the system it leaves it as soon as possible, and avoid that half-finished jobs accumulate in the system. Nevertheless, the system does not guarantee that the jobs are completed in the same order as they were submitted.

As said, the master tries to distribute task instances over the available workers as efficiently as possible. Since scheduling decisions cannot be retracted, and since the performance information about a worker might be imprecise or outdated, masters only submit a new job to a worker if it has an empty queue. The extra task reduces the idleness of the node without over-committing to the node. The master decides whether a queue is empty based on knowledge provided by the worker, and based on the number of outstanding jobs sent by the master itself.

Although the scheduling decisions in Maestro are made by the master, in some sense it uses is a work-stealing approach, since a worker that advertises that is has an empty queue is more likely to get a new job. Also note that the system can function properly without recent knowledge about all nodes. Stale or non-existent information about some nodes may cause suboptimal scheduling decisions, but the system is still able to function.

While honoring these limits, the master tries to select the fastest worker for a task. More precisely, it selects for each task the worker that is likely to have room for a new task and that provides the shortest estimated *makespan*: the completion time for all the remaining computations in the job, plus all necessary transmission and queuing times. For this purpose, every node keeps track of the following parameters:

- For every type of task and every known node, the transmission time of the input data of each task from the node to each node. We represent the transmission time from master node a to worker node b for the data of task type w as $t_{w,a,b}$.
- For every type of task, the number of task instances in the worker queue of the node. The number of task instances of task type w on worker a is represented as $n_{w,a}$.
- For every type of task, the queue time per task in the worker queue. The queue time for task type w on node a is represented as $q_{w,a}$.
- For every type of task, the execution time of a task type on the node. The execution time of task type w on node a is represented as $x_{w,a}$.
- For every type of task, the makespan of a task type on the node from the moment it enters the worker queue. The makespan of task type w on node a is represented as $N_{w,a}$.

The first parameter is specific for a particular pair of nodes. It is computed as a decaying average of the transmission time of the most recent task instances. To let the master measure the transmission time, the worker sends a 'task received' message to the master as soon as it has received the task. The master uses the total 'turnaround' time of this exchange as the estimated transmission

time, but the resulting over-estimate of the transmission time is arguably beneficial, since significant overhead in this area is a reason to avoid the node.[1]

The other four parameters are computed by the node, and are distributed to all nodes. Whenever one of these parameters changes, nodes that have recently submitted tasks to the worker are immediately updated, while all other nodes are frequently updated through a gossiping mechanism. Note that we assume that the performance of previous tasks predicts the performance of future tasks. This is only true if the tasks have repeatable execution times, so for optimal performance of the system this assumption should hold.

Using the parameters listed above, each node selects a worker for each task. Only workers that have fewer tasks outstanding than their allowance are considered. From these workers, the one that has the shortest estimated makespan is selected. The makespan is computed as follows: For a job consisting of tasks $w_0, w_1, \ldots w_n$, the makespan on node a starting with the transmission of task w_i to a node b is

$$M_{w_i,a,b} = t_{w_i,a,b} + n_{w_i,b} \cdot q_{w_i,b} + x_{w_i,b} + N_{w_{i+1},b}$$

while the makespan starting with the entry of a job of type w_i in the master queue is:

$$N_{w_i,a} = \begin{cases} 0 & \textit{for } i \geq n \\ r_i + \min_{b=0\ldots m}\ M_{w_i,a,b} & \textit{otherwise} \end{cases}$$

where r_i is the queue time on the master for a job w_i. This parameter is only maintained locally, and not transmitted to other nodes. Note that in this computation we ignore that a worker might be busy at the moment. This is because the limitations on worker queue size are not intended to influence long-term node selection.

As said, the set of nodes may grow or shrink during the computation. Whenever a node joins the computation, the underlying Ibis Portability Layer (IPL) reports this event [10]. Each Maestro node then adds the new node to the list of workers. The initial estimate for $t_{w,a,b}$ of each task is set very low to encourage the other nodes to send tasks to the new node and learn about its performance. The $x_{w,a}$ of the new node is computed by an estimator method of that is an optional part of a task implementation. The estimator typically runs a benchmark. If no estimator method is provided, a very low estimate is used that, again, encourages nodes to submit tasks to the node and learn about its performance. As always this performance information is distributed to the other nodes by gossiping.

Although the computation is most efficient when every worker supports all task types, Maestro allows particular tasks to be restricted to a subset of the nodes, for example because only these nodes can access a resource required for the task. This fits naturally in the parameter exchange framework: If a node w cannot handle a task a, it simply advertises its $x_{w,a}$ for the task as infinite, and it will never receive such a task.

Whenever a node leaves the computation, either in a controlled manner or because it crashes or becomes unreachable, all other nodes remove the node from their list of available workers, and put any outstanding work on the node back in their master queue. The IPL detects 'dead' nodes by sending regular test messages. If a node does not respond to the test messages for too long, it is declared dead, and this information is propagated to the remaining nodes. The detection is assisted by the Maestro communication routines, which report all nodes that trigger a communication timeout as potential dead nodes. Such nodes are then scheduled for an early test message.

[1]The alternative, directly measuring the transmission time, requires accurate synchronization between the clocks on the sending and receiving nodes, which would introduce a significant complication.

Maestro nodes are the most efficient if they have up-to-date information about the performance parameters of the other nodes. Therefore, workers send updates of their state to recent masters, and as part of their regular communication. Moreover, a separate gossiper on each node tries to ensure local performance information is efficiently distributed, and that the local information about other nodes is not stale. All performance information is timestamped, older information never overwrites newer information.

Each gossiper tries to keep the 'staleness' of its performance information below a fixed time interval. Whenever the information about a node becomes too old, an information request is sent to that node. Both the request and the reply contain gossip information. In fact, in the current implementation the entire collection of information is always sent. Therefore, a node may receive updated information about a node without having to send a request to it.

The gossiper engine is also subject to a communication quota. A message can only be sent if there is enough quota. The quota is regularly incremented. Local state updates, and the appearance and disappearance of nodes, also add quota to the gossiper.

The system described thusfar is entirely homogeneous: all nodes behave exactly the same. However, it is also necessary to insert work in the system, and administer the results of a job run. In the current Maestro implementation one node is elected to handle this, called the *Maestro* node, in analogy to the master in a master/worker system. This limitation is not fundamental; the system can easily be extended to support multiple Maestro nodes.

To increase the robustness of the system, the master of each node keeps sufficient information on each outstanding job to distribute it again if the node it was distributed to is reported to be dead or leaving. Moreover, whenever a master has an empty queue, it searches its list of outstanding jobs for jobs that are outstanding for a long time, and it distributes them again.

4. EVALUATION

Although the Maestro framework is designed to perform well on highly irregular and dynamic systems, we first evaluate its efficiency on a homogeneous system where we can predict the optimal performance, and compare Maestro performance against that. We then introduce a number of simple perturbations in the system, and evaluate the response of Maestro, and only then we show results on a real heterogeneous system. Since one of the benefits of our approach is that it is very robust against failures in the system, we will also evaluate that aspect. Finally, we evaluate the performance on a system with multiple clusters.

Since each Maestro node must learn the performance of every other node, and only part of this knowledge can be shared between nodes, we must expect some learning overhead. We must also expect that this is proportional with the number of nodes in the system. Although some of the learned knowledge is shared, in the worst case each node must try every inefficient node and communication link before it can take their poor performance into account. A large part of this learning is done at startup, so we will also examine the difference in efficiency of a Maestro system in the startup phase and later in the run.

We do not compare our system with other dataflow frameworks. The peer-to-peer nature of our approach is unique, which makes comparison hard. Moreover, the purpose of this work is not to show that a more efficient scheduler can be constructed that existing ones, but rather that a peer-to-peer approach can be used for an efficient and robust system. Also note that in the 'onejob' benchmarks we show below, only the master of the 'maestro' node has jobs to distribute, so that benchmark is representative of centralized master/worker systems.

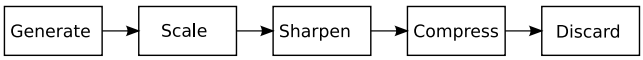

Figure 3: The computational flow of the benchmark program.

4.1 Setup

Unless stated otherwise, all measurements we show are the result of a run of at least 5 minutes. They reflect the time that is required for the entire run of the benchmark, including startup and termination. We have not done a formal statistical analysis of these measurements, but in our experience they are repeatable to within 5 to 10%.

Unless stated otherwise, the measurements were done on the 'VU' cluster of the DAS3 system[2], a cluster with 85 nodes, where each node has two dual-core 2.4 GHz AMD Opteron processors. Thus, each node has 4 cores available. The nodes are interconnected through a Myrinet network. In two experiments all five DAS3 clusters are used: the other clusters are similar to the 'VU' cluster, but on some of the nodes have two single-core processors, and one cluster has no Myrinet communication links but only 1GB Ethernet.

For our benchmarks we use a sequence of video processing steps similar to what might be used for 'up-conversion' of DVD movies to a higher resolution, see Fig. 3. Each frame in the video is scaled up from 720×576 to 1440×1152 pixels, sharpened with a 3×3 convolution filter, and converted to a JPEG image. In a production environment the input images would be read from disk, and written back to disk after processing, but the disk I/O might influence our measurements. Instead, we generate the input images ourselves as the first stage in the computational flow, and discard the output images without writing them to disk (also a separate step).

4.2 Homogeneous systems

On a homogeneous system the most efficient way to execute this computation is to execute all tasks of each job on the same node, since nothing can be gained by moving to another node. This way the communication cost between processing steps is minimal. Also, the jobs should be distributed evenly over the available nodes, except that the load of the master node must be reduced, since it has additional tasks. Obviously, Maestro should be able to discover this configuration itself.

If we know beforehand that the system is homogeneous there is little point in using the elaborate scheduling mechanisms of Maestro, and there is even little point in separating the computation into separate tasks, since this can only introduce overhead. However, this configuration gives a good baseline for the more complicated configurations, and makes it easy to evaluate the overhead of the Maestro scheduling mechanisms. Figure 4 shows the distribution of the tasks over the nodes for a homogeneous system with 20 nodes and 40000 frames as input. Each frame is computed in a separate job. In each stack in Fig. 4 the block height is proportional with the number of frames handled by a processor. In each stack the nodes are placed in the same order. To increase the readability of the graph, the results are sorted to place nodes with roughly the same number of executed 'Scale' tasks together (this will be used in subsequent experiments). Within each group the nodes with the smallest variation in frame counts between the tasks are placed at the bottom. As the figure shows, the self-organization of the system is working: with few exceptions, the tasks are distributed evenly over the nodes. Also, the number of tasks remains roughly constant from one step in the flow to the next, indicating that there is little communication between the nodes.

[2]See `www.cs.vu.nl/das3`.

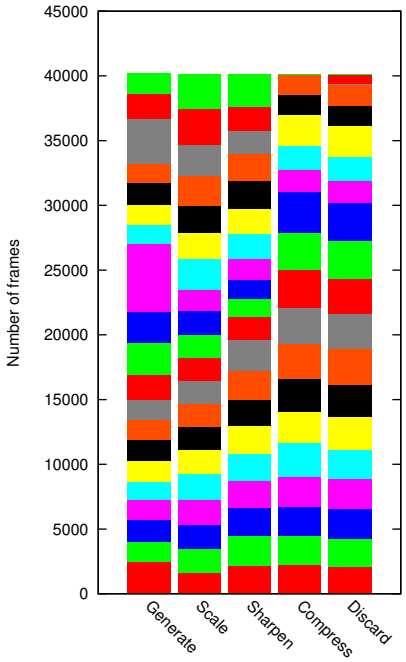

Figure 4: The distribution over the nodes for each of the tasks in the benchmark flow for a run with 20 homogeneous nodes and 40000 frames. Each block represents the number of frames that were handled by a particular processor for a particular task. For each task the blocks for the different processors are shown in the same order. Thus, the blocks at the bottom of all stacks are for the same processor. The blocks above that are for one other processor, and so on.

In Fig. 5, the curve labeled Homogeneous nodes plots the performance of the system for different numbers of nodes. We express the performance as the required execution time per frame averaged over the entire benchmark run, including startup and shutdown. Thus, the lower the execution time per frame, the better the performance. The total number of frames in a benchmark run is 2000 times the number of nodes in the system, resulting in all cases in a run of about 400 to 600 seconds. The run shown in Fig. 4 is one point in this curve.

For reference, the curve Homogeneous nodes, one task in Fig. 5 plots the results for a similar setup where all steps of the computation are implemented as a single task. Here, Maestro is used as a traditional master/worker system. This version does not have the overhead of handling the individual tasks, and is not subject to scheduling decisions on the individual tasks in the computation. As Fig. 5 shows, the overhead of using separate tasks is about 15% for small numbers of nodes, increasing to nearly 25% for larger numbers of nodes. Thus, although Maestro is unnecessary for a small homogeneous system, it is able to use such a system fairly efficiently.

4.3 Disturbances

To study the behaviour of Maestro for more complex systems, we use the same setup as before, but introduce a number of artificial disturbances. First, we disable the execution of the scaling task

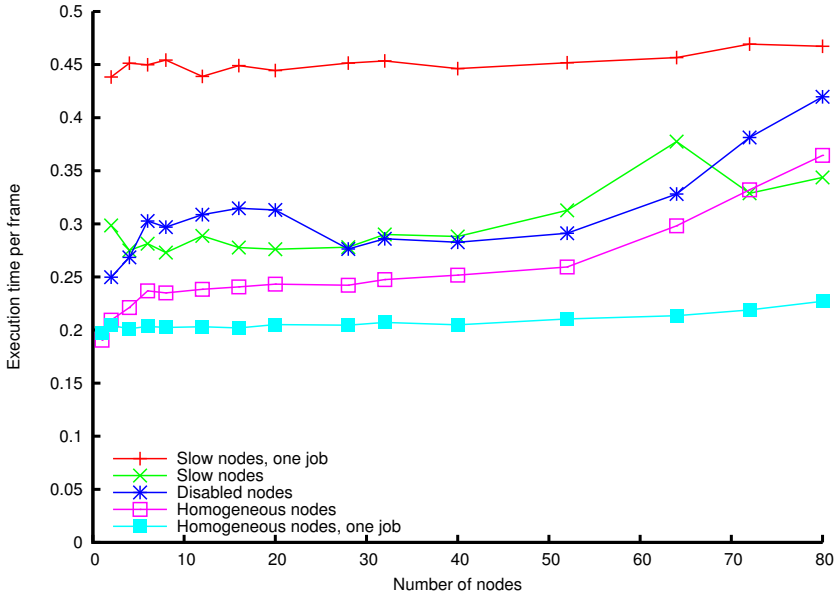

Figure 5: Execution time in seconds per frame for the computational flow of Fig. 3 for various numbers of nodes and different implementations of the flow. In non-homogeneous configurations half the nodes execute the 'scale' step 4 times slower or not at all, and the other half of the nodes executes the 'sharpen' step 4 times slower or not at all.

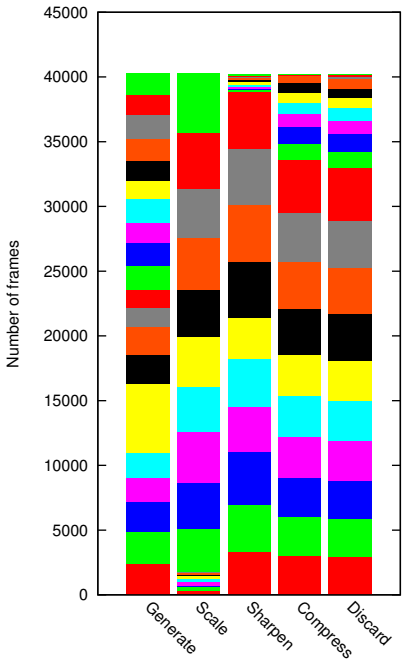

Figure 6: The distribution over the nodes for each of the tasks for a run with 10 nodes with slow scaling, and 10 nodes with slow sharpening. As the distributions show, for both these steps the slow nodes are avoided.

on half of the nodes, and the sharpening task on the other half. The system is now forced to use more than one node to implement the flow, with inevitable overhead. The resulting performance is plotted in Fig. 5 in the Disabled nodes curve. As the curve shows, there is some additional overhead, but the Maestro nodes are still able to use the system efficiently.

In the next experiment, we artificially slow down the scaling task on half of the nodes by executing the task four times instead of once. Similarly, the other half executes the sharpening task four times slower. Ideally, the Maestro nodes should perform at least as well as in the previous configuration, since they could decide to avoid the slow task processing, like they were forced to do in the previous experiment. The results are plotted in the Slow nodes curve of Fig. 5. As these results indicate, the Maestro nodes are indeed able to find an equally efficient configuration for this case. In analogy to Fig. 4, Fig. 6 shows the distribution of the tasks over 10 nodes with slow scaling and 10 nodes with slow sharpening for a run with 40000 frames. The plot clearly shows that due to the self-organization half the nodes are almost entirely avoided for the Scale step, while the other half is avoided for the Sharpen step.

To evaluate the performance of Maestro on nested parallel jobs, we replace the scaling task with a task that splits an image into 9 stripes of equal size, starts a nested job to apply scaling on each stripe, and combines the result into one large image. The results are plotted in Fig. 7, together with a number of curves from Fig. 5 for reference. As the experiments show, for small numbers of nodes the overhead of the extra computations and administration is low, but in the current implementation of Maestro this variation is inefficient for systems with a large number of nodes. The exact cause of this inefficiency must still be determined.

Finally, the Slow nodes, one task curve in Fig. 5 shows the results when the computation is performed as a single task, and again half the nodes are four times slower for sharpening, and the other half is four times slower for scaling. Since the nodes can now no longer avoid using the slow sharpening and scaling, there is a

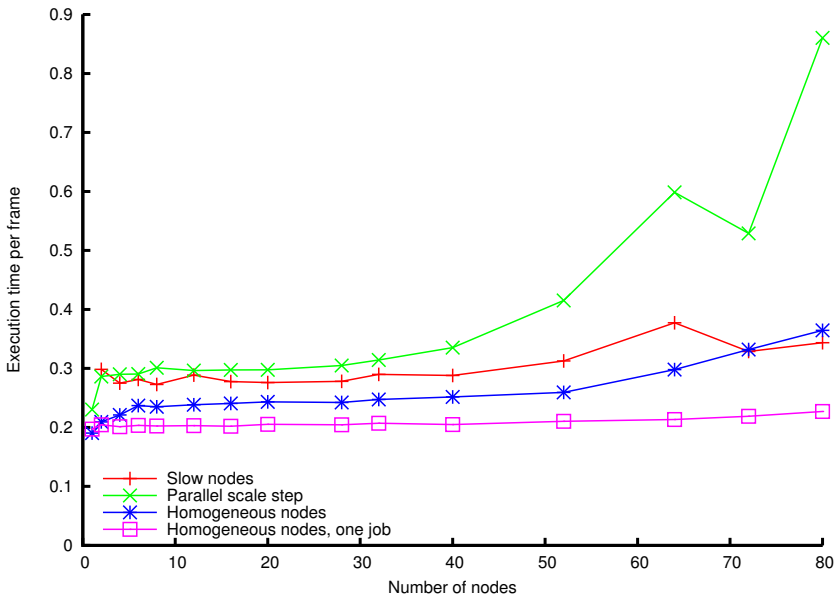

Figure 7: Execution time in seconds per frame for the computational flow of Fig. 3 for a computation with a parallel 'scale' step. For reference a number of results from Fig. 7 are repeated.

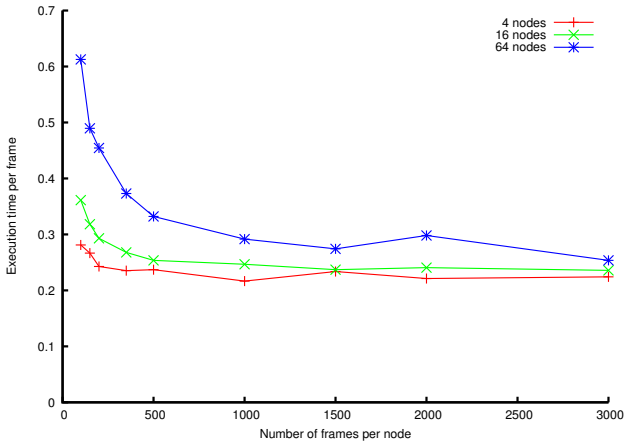

Figure 8: Execution time per frame for different numbers of frames for different numbers of nodes for systems with homogeneous nodes. This illustrates the learning effect of the Maestro nodes. See the text for the interpretation of this curve.

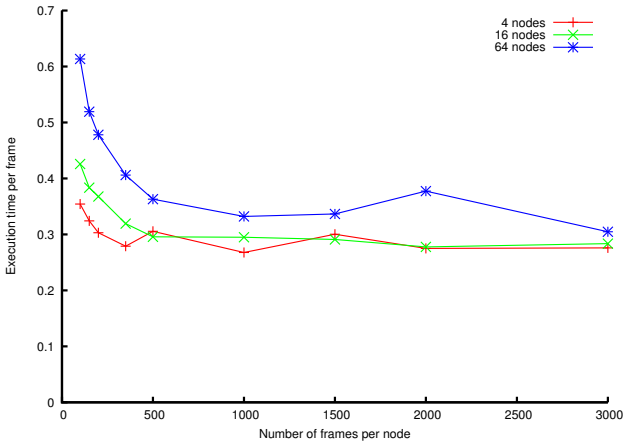

Figure 9: Similar to Fig. 8, execution time per frame for different numbers of frames for different numbers of nodes for systems with nodes with a slow scaling or sharpening step. This illustrates the learning effect of the Maestro nodes.

significant performance degradation. Separating the computation in different steps is clearly an advantage.

4.4 Learning time

It is also interesting to examine how long it takes for the nodes to reach an efficient state. Since at startup each node runs a small benchmark to estimate $x_{a,b}$ for each step, the gossip engine will quickly distribute a good approximation of that parameter to all nodes. However, other parameters must be learned by the nodes, resulting in poorer performance for the initial frames.

The most obvious way to study this learning effect would be to plot the computation time per frame over time, but in a distributed system it is difficult to collect this information and correlate the results of the different nodes. Instead, we simply plot the execution time for runs with increasing numbers of frames to compute. A system that learns will show a lower average execution time for longer runs because it can use the learned efficient configuration for more frames.

Figure 8 shows the average execution time over the entire run for different numbers of nodes in a homogeneous system. Note that the results for low frame counts are inherently 'noisy', since the startup overhead is not averaged out. In Fig. 9 we plot the results for a similar experiment, where half the nodes execute the sharpening step four times slower, and the other half executes the scaling step four

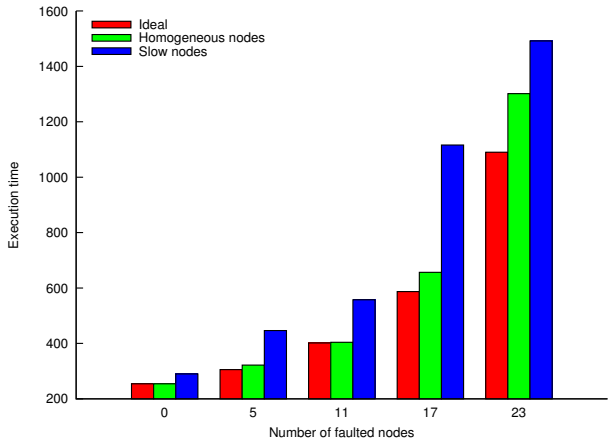

Figure 10: Execution time in seconds for homogeneous and non-homogeneous sets of nodes for a run of 30 nodes and 30000 frames, where different numbers of nodes are terminated prematurely. The Ideal bars plot the ideal extrapolation from the execution time for a homogeneous run without any terminated nodes.

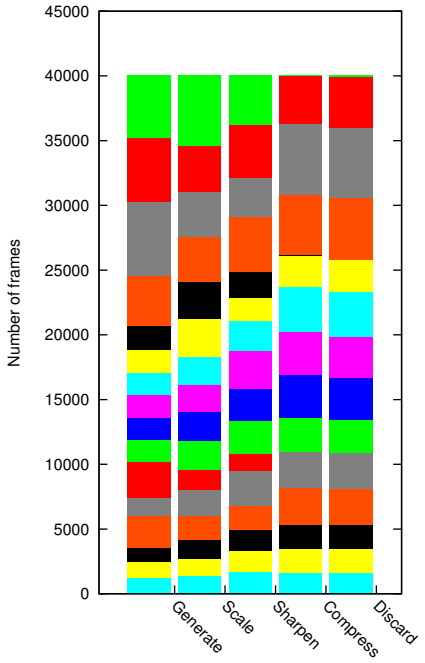

Figure 11: The distribution over the nodes for each of the tasks for a run on 5 different clusters of the DAS3 system with 4 nodes on each cluster.

times slower. The initial inefficiencies are clearly visible for runs with small numbers of frames, but the initial extra cost is already insignificant over a run of 500 to 1000 frames. As expected, runs with larger numbers of nodes require longer to learn, but even for large runs the initial learning time is small.

4.5 Fault tolerance

To examine the fault-tolerance of the of the system, we also did some runs where we terminated a fraction of the nodes early in the run. All these experiments were done on 30 nodes for 30000 frames, and for the homogeneous and non-homogeneous configurations shown before. After an initial delay of two seconds, every two seconds a random node was terminated, until the required number of nodes had been reached. The maestro node was excepted from this process, since this would require handling partial results of the application, and making the benchmark application itself fault-tolerant is beyond the scope of the project. Each terminated node only printed a message to a logfile and then stopped immediately. No attempt was made to gracefully withdraw from the computation, so all tasks on these nodes are lost without warning. This experiment obviously evaluates how well the system is able to detect and recover from faults. Moreover, it also evaluates the ability of the system to cope with changing circumstances: each terminated node not only introduces computations that must be redone, but also reduces the available processing power.

The results are plotted in Fig. 10, together with the ideal performance extrapolated from the performance of the homogeneous run without any node termination. The homogeneous configuration performs nearly ideally. The slow configuration does not fare so well when 23 of the nodes are terminated, but since only 7 nodes remain to complete the computation, there is an unavoidable and significant imbalance. For example, in this particular run 3 of the remaining nodes were slow at sharpening, and 4 were slow at scaling. Since the nodes to terminate are selected randomly, other runs may have another imbalance.

Clearly the system was able to complete the computation with little or no overhead despite the disturbances and loss of data associated with the terminating nodes.

4.6 Heterogeneous systems

Finally, we show some results for two runs on a truly heterogeneous system. The DAS3 cluster that was used for most of the experiments is used again, but the other four clusters of the DAS3 system are now also used. As explained, each of these clusters has some variations in its configuration. For example, some have only two processors per node, and one cluster has only 1GB Ethernet as communication network. Moreover, since the clusters are located at different sites in the Netherlands, the communication latency between some nodes is higher.

Figure 11 shows the results for 5 clusters with 4 nodes each without any artificial delay; Figure 12 shows a similar setup with the usual artificial slowdown in the scaling and sharpening steps. As these plots show, there is more variation between the contribution of the nodes. This expected, because the nodes are inherently inhomogeneous. Nevertheless, each processor is able to make a significant contribution. Thus, the system is still able to efficiently use all nodes.

5. CONCLUSIONS, FUTURE RESEARCH

We have shown in this paper that a peer-to-peer system using reinforcement learning is a robust way to implement dataflow computations on heterogeneous systems. With a remarkably simple task distribution approach, nodes can be used efficiently, even if they are heterogeneous, failure-prone, and linked with high-latency connections. This allows Maestro to run dataflow computations on systems that otherwise would be too complicated to use efficiently.

In our experiments Maestro was able to deal well with differences in performance, changing circumstances, differences in latency, and with faulty nodes; even in an experiment where 80% of the nodes failed, the system was able to complete the computation with little or no loss of efficiency.

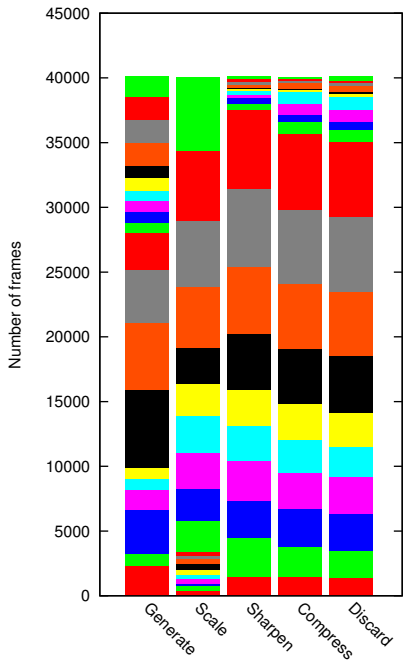

Figure 12: The distribution over the nodes for each of the tasks for a run on 5 different clusters of the DAS3 system with 4 artificially slowed nodes on each cluster.

For the current implementation we have mainly concentrated on the efficient execution of task sequences. Our experiment with a nested parallel step showed that for small numbers of nodes such a step was also efficient, but efficient implementation for large numbers of nodes requires further refinements of the system.

Despite these encouraging results, there are a number of areas where we expect that a more refined implementation would perform significantly better. First, the current system uses a fairly simple model to predict the performance of each node. A more detailed statistical model of the node performance, and a good treatment of the estimated errors in the estimates, would allow better scheduling decisions to be made. This way a Maestro master could base its decision on precise knowledge if it is available, or take the uncertainty of the estimated performance into account if necessary. In particular, a master could defer the scheduling of a task to wait for a fast node if it is confident enough in its estimate, instead of opting for a slower node as is done now.

Second, the method to gossip information is fairly simple, and we expect that a more sophisticated gossiper could reduce the amount of communication significantly. Similarly, more careful administration of state changes in each node could allow the system to only gossip information if there are (significant) changes.

Another refinement would be to let idle nodes run duplicates of tasks, so that their performance parameters are updated and there is some redundancy in the system. The obvious danger of this approach is that it may commit a processor to less important work when later it is better used for more urgent work.

Finally, the system could be generalized to support other performance goals. At the moment the system optimizes for the shortest makespan of each job. A different optimization criterion could be to minimize the total completion time of the entire run, to minimize the expended processor time, or even to minimize the total financial expenditure for the run.

6. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, pages 277–289, Asilomar, CA, Jan. 2005.

[2] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *Proc. of 5th IEEE/ACM International Workshop on Grid Computing*, pages 388–395, Pittsburgh, USA, Nov. 2004.

[3] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-centric allocation of independent tasks on heterogeneous platforms. In *International Parallel and Distributed Processing Symposium (IPDPS 2002)*, pages 67–72, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[4] J. A. Boyan and M. L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. In *Advances in Neural Information Processing Systems 6*, pages 671–678. Morgan Kaufmann, 1994.

[5] R. Buyya. Grid economy comes of age: Emerging gridbus tools for service-oriented cluster and grid computing. In *P2P '02: Proceedings of the Second International Conference on Peer-to-Peer Computing*, page 13, Washington, DC, USA, 2002. IEEE Computer Society.

[6] R. Buyya, J. Giddy, and H. Stockinger. Economic models for resource management and scheduling in grid computing. *Concurrency and Computation: Practice and Experience*, 14:1507–1542, 2002.

[7] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. Gridflow: Workflow management for grid computing. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 198, Washington, DC, USA, 2003. IEEE Computer Society.

[8] Cardif University. the Triana project. webpage, 2003. url: `http://www.trianacode.org`.

[9] A. Chakravarti, G. Baumgartner, and M. Lauria. Application-specific scheduling for the organic grid. *Cluster Computing, IEEE International Conference on*, 0:483, 2004.

[10] N. Drost, R. van Nieuwpoort, J. Maassen, and H. E. Bal. Resource tracking in parallel and distributed applications. In *Proceedings of the 17th IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, pages 221–222, Boston, MA, USA, June 2008.

[11] A. G. Escribano. *Synchronization Architecture in Parallel Programming Models*. PhD thesis, Dept. Informática, University of Valladolid, July 2003. Available at www.infor.uva.es/∼arturo/PhD/PhD.html.

[12] IBM. Exploratory stream processing systems. webpage, 2008. url: `http://domino.research.ibm.com/comm/research_projects.nsf/pages/esps.index.html`.

[13] C. Kelleher and R. Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137, 2005.

[14] G. Lee and J. Morris. Dataflow Java: Implicitly parallel Java. In *ACAC '00: Proceedings of the 5th Australasian Computer Architecture Conference*, page 42, Washington, DC, USA, 2000. IEEE Computer Society.

[15] M. Lorch and D. Kafura. Symphony - a Java-based

composition and manipulation framework for computational grids. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:136, 2002.

[16] R. V. v. Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *Proc. Eight ACM SIGPLAN Symp. on Princ. and Practice of Par. Progr. (PPoPP)*, pages 34–43, Snowbird, UT, USA, June 2001.

[17] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[18] K. v. Reeuwijk, R. V. v. Nieuwpoort, and H. E. Bal. Developing Java grid applications with Ibis. In *Proc. of the 11th International Euro-Par Conference*, pages 411–420, Lisbon, Portugal, September 2005.

[19] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. Streamflex: high-throughput stream programming in Java. *SIGPLAN Not.*, 42(10):211–228, 2007.

[20] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[21] R. van Nieuwpoort, T. Kielmann, and H. Bal. User-friendly and reliable grid computing based on imperfect middleware. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC'07)*, Reno, NV, USA, Nov. 2007.

[22] C. J. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[23] H. F. Wedde and M. Farooq. A comprehensive review of nature inspired routing algorithms for fixed telecommunication networks. *J. Syst. Archit.*, 52(8):461–484, 2006.

[24] P. G. Whiting and R. S. Pascoe. A history of data-flow languages. *IEEE Annals of the History of Computing*, 16(4):38–59, 1994.