ABSTRACT

There is an increasing demand for performing expensive computations on large volumes of data, making the use of parallel and distributed platforms a natural approach for addressing these requirements. The current trend while building such platforms is to employ multiple machines containing multiple processors, each processor comprising multiple cores, which allows the exploitation of multiple levels of parallelism. However, developing efficient and scalable applications that exploit efficiently those levels remain a challenge, because both the application requirements are very diverse, and a large portion of the applications is irregular, so that their characteristics and requirements change within a single execution. In this paper we present Anthill, a runtime framework that supports scalable and efficient parallelization of a wide range of applications on a variety of parallel platforms. We describe the foundations and two implementations of Anthill, and also the steps to parallelize applications within the framework. We also present several experimental results of the parallelization of relevant applications, where we achieved almost linear speedups using up to hundreds of processing cores.

1. INTRODUCTION

Many current computer applications involve expensive computations on very large data. At the same time, demand for interactive response times is very high, which creates a demand for compute power way beyond that of a single computer. Fueled by the continuous drop of cost per unit and increase in the power of the shelf components, the current trend in high performance computing is to make use of very large sets of machines with fast networks interconnects. On the other hand, the latest developments in the micro-processor arena have brought multiprocessing into all the newest CPU chips. Those chips, when used in the machines of large clusters and other computer aggregates, lead to multiple levels of parallelism in the system. To take full advantage of such systems, applications must be built with parallelism in mind both at the inter-machine level and at the single machine level.

Programming efficient parallel systems, in particular for such heterogeneous environments, remains a complex problem. One common environment used for that is the combination of MPI [1] (for inter-machine parallelism) with POSIX threads (for intra-machine/multi-core parallelism). While a number of computer scientists demonstrate the capacity of such environments to achieve high computing rates for many applications, as computers are becoming a widely used tool in many fields, such task are more and more laid on the shoulders of less experienced programmers. These are the application domain specialists, the actual originators of the demand for high-performance computing. Bridging the programming gap that arises has, therefore, become a central issue in computer science.

Several programming frameworks and middlewares have been proposed aiming to ease the development and efficient execution of parallel applications. Several of them, however, assume dedicated resources, fixed resource allocation strategies or a very simplified application semantics (embarrassingly parallel), failing to provide the desired performance that applications expect from large pools of resources.

In this paper we describe Anthill [9], a runtime framework which makes it possible for a broad class of applications to take advantage of parallelism in loosely coupled, heterogeneous environments. It allows parallelization of applications in several dimensions by partitioning the applications into a set of filters that communicate using streams (Filter-Stream model). These filters are then instantiated on
the distributed platform for execution, allowing multiple replicas of each filter to be created transparently. Previous results have shown that efficient and scalable implementations of applications in domains ranging from complex parallel image processing [18, 12] to parallel data mining [9, 20, 2], can be achieved using Anthill [9].

The paper is organized as follows. Section 2 presents a description of Anthill basic components. Following that, in Section 3 a higher level view of the programming and run-time environments exposed by Anthill is described. Section 4 presents several performance results obtained by applications implemented and Anthill. Some related work is discussed in Section 5 and we conclude in Section 6.

2. ARCHITECTURE OF ANTHILL

Anthill is a runtime framework inspired by the filter-stream programming model of DataCutter [3, 4]. This model uses a data-flow approach to decompose the applications into a set of processing units referred to as filters. Data is transferred through the filters using communication streams which allow fixed-size untyped data buffers to be transferred from one filter to the next.

The application decomposition process leads to the possibility of task parallelism at run-time, as the application becomes a set of filters connected like a directed graph. At execution-time, multiple (transparent) copies of each of the filters that compose the application are instantiated on several machines of the system and the streams are connected from sources to destinations. The transparent copy mechanism allows any vertex of the application’s graph to be replicated over many nodes of the execution platform and the data that goes through each filter may be partitioned among the copies creating data parallelism, as shown in Figure 1.

Anthill, therefore, tries to exploit the maximum parallelism in applications by using all three possibilities discussed above: task parallelism, data parallelism and asynchrony. By dividing the computation into multiple pipeline stages, each one replicated multiple times, we can have a very fine-grained parallelism and, since all this is happening asynchronously, the execution is mostly bottleneck free.

We now present a more detailed description of the several components of Anthill.

2.1 Basic Run-Time Framework

The implementation of Anthill consists of a library that is available for application developers to build their applications and a daemon that runs on each of the available compute nodes. There are, currently two implementations of Anthill. One based on the PVM [?] and another one, still experimental, that uses MPI [?]. An Anthill application is composed of three major parts: The application console, the configuration file and the filter library. For the application’s point of view, both Anthill implementations are identical.

The application console is a simple executable program which invokes a library function to initialize the Anthill environment. This function receives the configuration file as an argument and instantiates the filters on the compute nodes as defined by that description. It then creates one or more works which are the parameters for the filters to start executing. Before exiting, the console also calls a function to shutdown the entire Anthill environment. A simple console program example is shown in Figure 2.

```
#include "anthill.h"

int main(argc, char *argv[]) {
    int myWork = 1;
    Layout *systemLayout = initAh("myconf.xml", argc, argv);
    ... appendWork(systemLayout, (void *)&myWork, sizeof(int));
    ... finalizeAh(systemLayout);
    return 0;
}
```

Figure 2: A Sample Application Console.

The configuration file is an XML description of the application. An example is shown in Figure 3, in which we can see it is composed of three sections. The first one, is the specification of the available compute nodes. The second describes the application filters instantiation on the preceding computer nodes. And finally, in the third section, it describes the filter interconnection.

In the host description section, each machine is listed by hostname. Associated to each machine, there may be a set of resources that are declared by the application developer. The semantics of these resources is defined by the developer and is used later on to make decisions about the placement of the filters.

The filter placement section lists the filters that comprise the application and the placement for them on the available nodes, which can be implicit or explicit. The application developer can specify the number of instances of each filter that needs to be created at execution time as well as where each of the instances need to be placed. The placement can specify the hosts themselves, or can use

For several applications, the natural decomposition is a cyclic graph, where the execution consists of multiple iterations over the filters. An application starts with data representing an initial set of possible solutions and as these pass through the filters, new candidate solutions are created. Those, in turn, have to be passed through the network so they can also be processed. In our experience with developing applications in Anthill we noticed this behavior also leads to asynchronous executions, in the sense that several candidate solutions (possibly from different iterations) are processed simultaneously at run-time. This is similar to software pipelining and the unrolling of a parallel loop.
the resources that have been defined in the host description section. If more instances are requested than hosts are explicitly assigned, the remaining instances are allocated randomly on the hosts. A weight can be specified as a host resource, which determines the likelihood of a node being selected to execute a filter instance.

The last section of the configuration file defines the streams. Notice on Figure 3 that the streams are defined as unidirectional data pipes by the subsections from and to in the XML description. Also notice that there is no mention of multiple instances of the filters, rather they are connect as if they are logical units of processing. The multiple instances, or the transparent copy mechanism is handled internally by Anthill. Two additional attributes are available to the stream declaration which allows to control the policies Anthill will use, when a stream is connecting multiple filter instances. Notice also that each stream specifies a port name. That allows filters to communicate using multiple input and output streams. They are identified by this port name in the filter implementation.

Finally, the filters in Anthill are defined by a dynamic link library. While in DataCutter, the filters where specified by extending a C++ virtual class, in Anthill, the developer is expected to provide a dynamic-link library (DLL) for each filter with the implementation for the functions \texttt{Init}, \texttt{Process} and \texttt{Finalize}. The DLL is specified in the configuration file and loaded by the Anthill runtime system. After processing the placement section of the configuration file, the console sends messages to the application daemons running on the specified nodes and requests the loading of the appropriate filter libraries. After that, for each \textit{work} that the console submits to the filters, the three filter functions are invoked in turn. The first two functions receive as arguments a generic chunk of memory (a pointer and a size) that contains the \textit{work} as defined by the developer. This allows flexibility in the implementation of the filters and the passing of generic parameters.

As in all dynamic link libraries, two functions may also be available. One for initialization right after the library loading time. And a corresponding one for cleaning up as the library is removed from memory. Such functions may be useful to perform some expensive initializations that are not required to be re-executed in a case where multiple \textit{works} are sent to the filter. One example of such situation is for a filter that uses a Java Virtual Machine (JVM) for execution. The initialization of a JVM is an expensive operation, and time can be saved by doing it just once as opposed to for every \textit{work}.

2.2 Filters

The application developers are free to design their filters anyway they like. Anthill will invoke the three functions described above, but the implementation of those functions are up to the developer. A library is available that allows the usage of more of the Anthill resources.

A set of library functions allows the programmer to operate with the streams. As mentioned above, the streams have a port name associated to each, and are unidirectional. For the filter perspective, they can be either an input stream or an output stream. The programmer can obtain a stream descriptor by passing its port name to a library function. With the descriptor in hand, the user can write or read data to or from the stream, using appropriate library functions. The messages are contiguous bytes in memory. Anthill makes no assumptions about the messages.

Filters do not open streams as they are received from the underlying framework. However, if a filter decides that it no longer needs an output stream, it can invoke a function to close it. In effect, this function sends a special message of End-of-Work (EOW) down the stream. After that function call, the stream cannot be used to transfer data anymore. When a filter receives an EOW from its input stream, it means that one of the instances of the preceding filter has finished processing. Filters can use such mechanism to synchronize and to determine the end of the processing.

Additional library functions are available to retrieve the total number of transparent copies of the filter that are currently executing, and also to obtain the rank of a particular filter instance. These functions are used to allow the instances to figure out their positions in the global state of the filter, and determine their tasks within that global state.

Anthill also provide detailed instrumentation within its library functions allowing developers to identify bottlenecks of the application, extract of detailed performance of the execution, and it can be used dynamically as well, for automatic reconfiguration and other features.

2.3 Streams

The streams represent the communication abstraction among filters. Logically it is unidirectional pipe between two filters. At run time, however, several instances of each filter may be running simultaneously on the same or on different compute nodes. The message transfer abstracted by the stream is implemented using the underlying MPI or PVM system, depending on the version of Anthill that is running.

The original DataCutter [3, 5] had basically three policies for delivering a message to the filter instances. In the first basic case, the new message received by the stream is delivered to the next instance of the destination in a round-robin fashion. This policy is suitable for applications in which the processing time of the messages on any of the instances is very regular. To account for differences in the node performance, a weighted round-robin policy is also implemented which delivers more messages to faster filters.

Figure 3: A Sample Configuration File.
proportionally to their statically defined, relative processing times. When the application, however, has the characteristic of being irregular, meaning that the processing time of a particular message depends on the message itself, a load imbalance can generate a significant drop in performance of the application. For such cases, the developer can use a demand-driven delivery policy which keeps track of the filter instances availability. Each message is delivered to the next idle filter. There are overheads associated with such a policy, but for cases where the load imbalance is severe, the overheads are offset by the benefits.

Although these three basic stream policies are generic and sufficient for implementing applications for heterogeneous systems, Anthill takes this into one step further and adds two more stream policies, broadcast and labeled-stream, that enables a totally new programming paradigm. Broadcast policy tells the runtime system that message should be delivered to all instances of the destination filter. The labeled-stream policy can be perceived as a selective multicast policy. For streams under that policy, the messages are associated with a label. And there is a function, also specified in the configuration file, that receives the label for each message flowing through the stream and returns a hash value which is used to select the set of destination instances that should receive that message, which is delivered appropriately.

The label stream policy has been used extensively in Anthill applications to allow filter instances to partition the original filter state while still maintaining linear speedups. In essence, the labeled stream policy guarantees locality of reference on the filter instances, guaranteeing that each will receive messages that update only state variables available locally.

3. THE RUN-TIME ABSTRACTION
In essence, Anthill provides a component based data flow model for running applications on distributed environments. Even though the model is not new and several other systems based on the same model are available in the literature, we believe that we inherited several unique features from the original DataCutter and improved upon it with some additional interesting features.

Some of the more interesting high level features available in Anthill are related to improving the programming abstractions for the application developers. While the transparent copy mechanism was already featured in DataCutter, when the filters had state, there were only two ways of implementing the filter: replicating the entire state and adding a global combine filter, or explicitly partitioning the state on the filters which would eliminate the transparent on the transparent copy mechanism.

Neither solution is satisfactory as both of them put additional burden on the programmer and/or incur additional overhead. The label stream policy resolves that problem in a generic way. The programmers need to define a label that associates a message to a state variable in the application, and the mechanism guarantees that all messages with the same label are delivered to the same filter instance. Anthill further raises the level of abstraction by creating a more flexible execution environment for the filters and simplifying the programming interface for them.

In this section we describe some of these features in Anthill.

3.1 Global State Manager and Dynamic Re-configuration
Anthill already provides a programming framework which facilitates state partition among instances of a filter. In addition to that, the Anthill framework also provides a global address space to allocate state variables for the filters. This global address space is partitioned across all nodes that share a particular state, and is provided by an additional daemon that executes along with the Anthill daemon.

This global state daemon can be seen as a distributed shared memory system. Except that it is custom designed for sharing memory across Anthill filter instances which have a very high locality of reference due to the label stream mechanism. This characteristic allows us to develop a low overhead daemon that provides Anthill with some migration capability. Application developers just allocate the state as if it is a global variable by using an interface very similar to the standard malloc/free library calls. The partitioning and eventual migration is handled transparently at run time.

Data migration can be necessary for two reasons. The initial state partition decided by the runtime system do not match the one generated by the hash function invoked at the label stream mechanism. This will cause at most one migration of each state variable during the execution of the application which was measured as a low overhead in our experiments. The second situation in which migration becomes necessary is if the filter placement is changed. At the moment, we see one reason for the placement to change: dynamic reconfiguration. As the execution of the application progresses, the runtime system can make a decision to change the placement of filters. More instances of a particular filter may become available, or other filters my experience a reduction in the number of available instances. This can be done to adjust for performance, or because more nodes become available for execution in a dynamic environment. In either case, Anthill can adjust itself at run time, without involving the application developers with extra programming overhead.

One additional feature enabled by such a setup is fault tolerance. The global state manager can maintain enough redundancy transparently to the user which would allow the application to keep running even if some of the compute nodes failed during execution. All checkpointing and eventual rollbacks can be entirely handled within the framework with no extra overhead to the application developer.

3.2 Filter Abstraction
The filter, as mentioned earlier, is the basic processing unit within Anthill framework. In principle, a filter is a thread of execution that invokes the three filter methods in a row for each work that is submitted. The developers are free to implement anything they want within those three functions. In addition, the developers are provided with a library that can be used to communicate using the streams that are connected as described in the configuration file for the application. Such a simple interface, while allowing maximum flexibility to the developers, also incurs in a few significant drawbacks.

First, our experiences with implementing applications for Anthill have shown that many consist of a main loop that propagates data down a processing chain. In Anthill, such applications map to a sequence of filters connected in a loop. Such an arrangement complicates the decision of the end of computation, in particular because there may be multiple transparent copies running of each filter.

Another drawback is that most of the filters actually consist of a
loop within the \textit{Process} function that receives some input from a set of streams, processes these inputs and then sends the outputs down another set of streams. Having these loops explicitly implemented by the developers puts on them the burden of dealing with data dependencies and synchronization issues that arise from the transparent copy mechanism.

And finally, a filter implemented as a big loop as described above will not easily benefit from the available hardware in the compute node, unless the developer put some substantial effort into it.

Anthill provides two higher level abstractions to address these drawbacks. The first abstraction is a automatic termination detection algorithm that runs within the library calls and is totally transparent to the user. The second is a event driven interface to the filters, which moves the abstraction more towards the standard data-flow model. Such abstraction also enables Anthill to make effective use of the underlying hardware, without additional burden to the application developer.

3.2.1 Termination Detection

As mentioned earlier, Anthill provides a function that executes the close of an output stream. The invocation of such a function cases the broadcast of a special token, called EOW to the destination filter instances. If the application interconnection graph is acyclic, it is easy to determine the program termination. Every acyclic graph has to have a source node, and that source node will eventually call the close function. That close will start propagating the information about the termination of that particular filter which will create another source node on the graph. And the process continues until all filter terminate.

When the application graph is cyclic, however, the problem becomes a lot more complicated. Delegating to the application developer to determine the termination is not accepted for it amounts to a distributed consensus protocol. Anthill, therefore, implements such an algorithm transparently to the users.

For the sake of the termination protocol, a filter instance on one side of a stream is connected to all instances of the filter on the other end of that stream. The algorithm works in rounds, when some instance suspects the computation is over. Each instance keeps a round counter \( R \) that is used by the protocol. A special process, called the process leader, is responsible for collecting information and reaching the final decision about whether or not to actually terminate the application.

Three types of messages are exchanged in the protocol: instances that suspect that termination was reached send \textit{SUSPECT} (\( R \)) to their neighbors (in both directions) stating they suspect termination in round \( R \); when an instance has received \textit{SUSPECT} (\( R \)) messages from all its neighbors, it notifies the process leader using a \textit{TERMINATE} (\( R \)) message, also identifying the round number; if the leader collects \textit{TERMINATE} messages from all filter instances for the same round it broadcasts an \textit{END} message back to them.

Besides the round counter, \( R \), each instance keeps a list of the neighbors which suspect the same termination round \( R \) has been reached. The core of the protocol is illustrated by the extended finite state machine in Figure 4. Each instance may be in one of two states: \textit{running} or \textit{suspecting termination}.

While an instance is computing and/or has data in its input streams still to be read it remains in the \textit{running} state and does not propagate messages for the termination protocol. If its has been idle for some time it moves to the \textit{suspecting termination} state and notifies all its neighbors by sending them a \textit{SUSPECT} message with its current round number. It keeps the list of suspected neighbors it collected while in \textit{running} state, since they were considered to be in the same termination round as itself.

In either state, if an instance receives a \textit{SUSPECT} (\( R' \)) message from another for its current round (\( R' = R \)) it adds the sender to its list of suspected instances; if it is for a newer round (\( R' > R \)) it clears that list before adding the sender (the only one known to be in that round so far) and updates \( R \). If it receives an application message from another instance, it removes the sender from its list of neighbors suspecting termination, since it is obviously computing. If the receiver was in the \textit{suspecting} state, it goes back to the \textit{computing} state, updates its round counter and clears the list of suspecting instances.

Whenever an instance in the \textit{suspecting termination} state has collected \textit{SUSPECT} messages from all its neighbors for a given round, there is a widespread suspicion that termination has been reached (although that may be true for just the vicinity of that instance, and not for all instances in the cycle). At that point the instance sends a \textit{TERMINATE} (\( R \)) message to the process leader. The instance remains in the \textit{suspecting} state, since termination has still to be confirmed by the leader.

The process leader, on its turn, must keep track of the newest termination round it has heard of (\( R \)). Whenever it receives a \textit{TERMINATE} (\( R' \)) message from a filter instance, it must compare \( R \) and \( R' \); if \( R' \) is lower, the message is simply discarded, since it relates to a round that is already known to have passed; if \( R' > R \), a new round has started and it is not relevant anymore, so the list of terminated instances must be cleared and just the sender of that message must be added to it; finally, if they are equal, another filter instance has joined the group of processes that suspect termination was reached, so it must be added to the list. When that list is complete with all processes the leader can declare that the application has ended. At that point it broadcasts the \textit{END} message and all processes take steps towards termination. In the filters instances, that causes the reinitialization of the termination protocol and the closing of the stream selected by the user with an end-of-stream notification.

3.2.2 Event-Oriented Anthill

To describe in simple terms, the idea here is to create a filter abstraction in which the explicit loop in the application filter code moves to an implicit loop within Anthill code. Such an arrange-
ment has several practical implications to the abstraction exposed by Anthill.

While the process-oriented filter abstraction required the programmer to implement a filter that may read from many input streams, now, the programmer has to implement a function that receives certain data elements and after processing them, generates output data elements. Handling multiple input streams force the programmer to resort to asynchronous I/O primitives and other data flow related issues in order to obtain the data required for processing. Problems like deadlocks or polling overheads are all eliminated as a burden to the programmer, which can now concentrate on the specific semantics of the application itself. With events, the programmer may indicate what should be done upon arrival of data on any stream, even with different actions for each source. The run-time system is in better condition to control non-blocking I/O, if necessary, and choose the right processing function for each event based on the programmer’s instructions. Such approach derives heavily from the message-oriented programming model pioneered by the x-kernel [16] and later extended to include explicit, user-defined events in Coyote [6].

Also, the process-oriented filter abstraction has no notion of intra-node parallelism. The filter is implemented as a sequential loop, and is executed like that. Developers may use different abstractions for exploiting multiple execution units within a filter, but that requires additional efforts and may create drawbacks by potential incompatibility of primitive used by different frameworks or of abstractions each of these additional frameworks provide or require. The event-oriented abstraction is not adding a different, conflicting abstraction, but actually closing an implementation gap between the model and the programming interface. A filter processes data as it flows through, possibly changing the nature of the data that is passed to the next filter. On those pretenses, a programmer, when defining a filter, should have to worry only with the transformation function, ran for each data unit in turn, and that may internally cause some data to be forwarded to the next filter. Deciding when such processing takes place, except for explicitly indicated data dependencies, is a task of the filter framework, not a concern of the programmer. Provided that data is available, dependencies are satisfied and there are available resources for execution, the function can be invoked. What is transparently accomplished by this is the effective unrolling of the original filter loop onto the multiple processing units available for computation. This feature is very useful for exploiting the full capability of the current multicore architectures without forcing the application programmers to keep all parallelism levels in mind explicitly.

We should emphasize the expressive power of this parallelization strategy. The developer, by using a single abstraction (the filter-stream model), with a simple programming interface (Anthill events), may create applications that exploit two-level parallelism (inter- and intra-machine, or cluster and processor), by just focusing on how filters communicate and what filters do upon the arrival of data.

4. RESULTS

In this section we present a set of experimental results to demonstrate Anthill performance responses for several of the different aspects discussed in this paper. We use three data mining algorithms as example applications.

- Eclat

A well-known algorithm to generate association rules. An association rule is an causality relation among frequent co-occurring items in a database. The first step in computing association rules is calculating all co-occurring sets that appear in database with frequency above a determined threshold. The Anthill implementation of Eclat [21] has three filters: generate candidates, merger, and adder. The first filter generates all candidates with size \( k + 1 \) from frequent item-sets with size \( k \). The merger filter calculates the local frequency of each candidate itemset generated. Finally, filter adder calculates the global frequency of each itemset adding local frequencies generated on previous filter.

- K-means

A clustering algorithm that groups objects regarding a user-defined similarity criteria [15]. It is based on the concept of centroids, which represent the objects that belong to each cluster. Each iteration of the algorithm assigns each object to the closest centroid, and updates its value properly. The K-means implementation using Anthill [18] has three filters: assigner, centroid calculator, and final. The assigner filter has the set of all objects the application is trying to cluster, as well as the initial set of centroids. It computes the distance of each object to all centroids, and assigns each object to the nearest centroid. The centroid calculator filter computes the new centroids based on the objects currently assigned to each of them. The algorithm terminates after a specified number of iterations or when no object changes its centroid after one iteration.

- KNN

K-Nearest Neighbors (KNN) [10] is a classical algorithm for classification. It is based upon the idea that the neighborhood of an object contains similar objects. Thus, we determine an object class by analyzing its neighborhood. Since it is difficult to determine its neighborhood, the algorithm uses the k-closest objects to define its class. The KNN algorithm in Anthill is divided in two filters: classifier, and merger. The classifier filter finds local k-closest objects of training database for each object in testing database. Then, it sends to merger filter, which receives the k-closest objects from each classifier instance, sort them and selects the global k-closest objects to assign one class to testing object.

4.1 Traditional Anthill

We used a synthetic database with size 2.24 GB for evaluating the performance the Anthill version of Eclat. This database mimics the transactions in a retailing environment. The experiments were performed with a minimum support value of 0.1%. This experiment used 32 GHz Pentium IV nodes with 1 GB main memory connected by Fast Ethernet, and running Linux 2.6. As can be seen in Figure 5, Eclat achieves super-linear speedup with 32 nodes.

Two synthetic databases were used to evaluate K-means implemented in Anthill, one with 800,000 and other with 400,000 points, each point with 50 dimensions. The speedup of the implementation is shown in Figure 6. It achieves almost a linear speedup for 400,000 points database and a super-linear speedup for 800,000 points. Both experiments executed in 24 nodes with the same configuration used on the Eclat experiments.

4.2 Reconfiguration
The reconfiguration strategy was evaluated using two statefull data mining applications: k-means and Apriori. In order to experiment with the reconfiguration model, the applications needed to be adapted to use our global state manager, discussed in Section 3.1, to store the distributed data. K-means has been modified in such a way that the objects being clustered are actually stored in the global memory and assumed to be part of the filter state. For Eclat, the database tuples had to be placed on the global state manager, as they may need to migrate in the case of reconfiguration.

For evaluating the reconfiguration, we designed the following set of experiments: We consider only cases in which more nodes become available. We choose three milestones of the application execution, discussed in Section 3.1, to store the distributed data. K-means has been modified in such a way that the objects being clustered are actually stored in the global memory and assumed to be part of the filter state. For Eclat, the database tuples had to be placed on the global state manager, as they may need to migrate in the case of reconfiguration.

Figure 7 shows the difference in execution times for the original and new implementations of each application. What we can see from both applications is that the overhead incurred by the global state manager are actually negligible. For the k-means algorithm, the average difference in the execution times for original and new implementation was around 1.5%, while for Apriori it is 1.8%. What the figure shows is that there is no significant penalty been paid by having the additional functionality as far as the application performance.

![Figure 5: Eclat speedup.](image1)

![Figure 6: K-means speedup.](image2)

![Figure 7: Execution times for the different implementations.](image3)

![Figure 8: Evaluation of different reconfiguration scenarios.](image4)

4.3 Multi-core Analysis

The first set of experiments (single machine) were run on a quad socket, dual core 2.6 GHz Opterons of 16 GB of main memory. The speedup experiments have been measured into a cluster of dual socket 2 dual core 2.6 GHz Opterons of 8 GB of main memory connected through a by 10 Gbps Infiniband.

Figure 9(a) shows the results of the single machine experiments. As can be observed, the results using one process/thread of the sequential, pthreads, and Anthill versions of the application are comparable and the overhead of the latter two versions are, respectively, 5.3% and 10.8%. As shown in the Figure 9(a), Anthill and
Pthreads implementations reduce the execution time as we increase the used resources, and the best speedups are achieved when we are using 8 threads in both frameworks, being 6.83 (Pthreads) and 6.80 (Anthill).

Figure 9(b) shows the experiments for the Anthill version of KNN using a distributed and shared memory environment. The speedups are calculated based on the sequential version of the algorithm. As the figure shows, the Anthill achieves a speedup of around 100 for 128 cores, delivering 78% of parallel efficiency.

5. RELATED WORK

Dean et. al, in [7] presented MapReduce, a system for massively parallel analysis of very large semi-structured data sets. The system model is expressed by two user defined functions: a map function that performs independent computation on stored data chunks, and the reduce that combines the results from the independent computations into a single result. The MapReduce programming model is similar to the filter-stream because it divides the application into processing stages that communicate over the network. The MapReduce framework also uses a storage system to exchange messages. While the MapReduce programming model is adequate for embarrassingly parallel applications, it is not suitable for many of the applications implemented using Anthill, which can be asynchronous and have fine-grain parallelism. An open-source Java-based implementation of MapReduce is Hadoop [19]. It is a distributed system based on Google’s MapReduce [8] and Google File System Infrastructure [11].

The Dryad [13] system is a framework which allows a programmer to use resources of distributed environments for running data-parallel programs. Programs in Dryad can be described as a graph, where processing stages are vertices and the communication channels are represented by the edges. The programming model provided by Dryad allows creation of application that can be mapped into a directed acyclic graph which subsumes other frameworks, such as Google’s map-reduce, or the relational algebra. Although interesting, the model provided by Dryad is also subsumed by the filter-stream programming that allows the creation of applications that are represented by a directed cyclic graph.

Ruoming Jin et al. have created FREERIDE, a middleware for implementing data mining applications which exploits environments with distributed and shared memories [14]. This framework can be used by a number of data mining algorithms as it computes generalized distributed reductions, a common operation for the class of algorithms. The authors also evaluate different strategies to synchronize data access and avoid errors in the reduction operation. Although the framework achieves good performance for the shown applications its programming model is not as generic as the one presented in this paper, since it can exploit shared and distributed memory systems for all applications that can be mapped in the filter-stream model, not only those based on reductions. For the applications considered, the speedups achieved with Anthill are linear on the number of available cores, which is not the case with FREERIDE.

6. CONCLUSIONS AND FUTURE DIRECTIONS

In this paper, we presented the Anthill run-time framework which supports the efficient execution of applications in heterogeneous distributed environments. Previous results have shown efficient and scalable implementations in domains ranging from complex parallel image processing [18, 12] (coarse-grain) to parallel data mining [9, 20, 2, 17] (fine-grain), can be achieved using Anthill [9].

We discussed the high level programming and execution abstractions provided by Anthill. Supported by our experimental results, we argue that the system provides a simple very high level interface for the application developers which can then focus on the application details, delegating the low level details to underlying infrastructure. At the same time, the system is capable of making effective use of the available hardware delivering high performance to the application.

One of the things that have not been address by existing runtime systems is support for heterogeneous cores. Anthill’s component-based architecture provides an ideal platform for such support. Application components (i.e., filters) that could take advantage of a specialized core can be easily instantiated on that core by Anthill runtime system. If filters have multiple implementations that could take advantage of different core-types, Anthill runtime system can also leverage that. Depending on the computational load and available resources Anthill can instantiate different implementations of a single filter on different core-types. Furthermore, by using stream abstraction, Anthill runtime system can take advantage of shared caches between cores or in-chip network. We are currently actively pursuing research in some of the directions.

Besides more aggressive support of the low level features of current architectures, another direction in which we are actively working is towards even higher levels of abstraction. In particular, we are investigating compiler techniques that can be leveraged for generating code for Anthill directly from sequential programs in high level languages. Several interesting aspects may arise from this re-
search. In particular if we consider that the granularity of the ideal application decomposition is tightly coupled to the available hardware resources. It might be useful to delegate program partition to a later time, when the system know what the execution environment will look like.

7. ADDITIONAL AUTHORS

8. REFERENCES