Biologically-Inspired Distributed Middleware Management for Stream Processing Systems

Geetika T. Lakshmanan and Robert E. Strom

IBM T. J. Watson Research Center, Hawthorne, NY 10532, USA
{gtlakshm,robstrom}@us.ibm.com

Abstract. We present a decentralized and dynamic biologically-inspired algorithm for placing dataflow graphs composed of stream processing tasks onto a distributed network of machines, while minimizing the end-to-end latency. Our algorithm responds on-the-fly to placement requests of new flow graphs or to modifications of an already running stream processing flow graph, and dynamically adapts to changes in performance characteristics such as message rates or service times as well as to changes in processor availability or link performance during runtime. Our algorithm is derived by analogy to pheromone-based cooperation between ants to fulfill goals such as food discovery. We have conducted extensive simulation experiments to show the scalability and adaptability of our algorithm.

Key words: Distributed Stream Processing, Task Placement, On-The-Fly Management, Biologically-Inspired, Self-Managing Middleware

1 Introduction

The complexity of current computer systems has motivated computer scientists to turn towards nature for inspiration. The fault tolerance properties and decentralized control achieved in natural systems combined with the intuitive simplicity of their design make them particularly appealing for solving problems in computer systems. Immune system architectures for computer security methods [1], firefly-inspired synchronicity for wireless sensor networks [2], bee colony behavior-based middleware platforms [3] and ant-based ad hoc network multicasting [4] are but a few examples of the ways in which biology has influenced the design of computer systems. Investigation of the collective foraging behavior of ants has sparked the field of *ant colony optimization* (ACO) algorithms [5]. These are probabilistic techniques for solving computational problems which can be reduced to finding optimal paths in graphs. Ant-based algorithms have been applied to solve combinatorial optimization problems such as the travelling salesman problem, quadratic assignment problem, graph coloring, job-shop scheduling, sequential ordering and vehicle routing [5]. In addition to these static problems, where characteristics do not change over time, ant-based algorithms have also been applied to stochastic time varying problems such as routing in telecommunications networks [6]. Given the adaptive capabilities built into ant-based algorithms, they are particularly well suited to such problems where solutions must be adapted online to changing conditions.

Adaptive online task placement is currently an important problem in distributed stream processing systems [7–13]. A distributed stream processing system (DSPS) streams data from multiple data sources or *producers* to multiple clients or *consumers* interested in the results derived from processing conducted on the data. Between the producer and consumer the data is processed through a number of computational tasks that are linked via a dataflow graph. Each task receives one or more streams of input data messages, either from a producer or from an upstream task, each message in a stream arriving asynchronously. In response to a message, a task performs a computation, which may access the message, read and modify internal state representing past history, and may or may not generate one or more messages which are sent either to consumers or to downstream tasks. Examples of tasks include operators, such as an incremental join, an aggregation, or a facial recognition operator. Flow graphs may be designed by an application designer, or may be separately specified by consumers who write queries. Each query defines the particular messages a particular consumer requires and the operators or computational tasks needed to derive these messages from producer streams.

There are a number of well known distributed stream processing systems in both academic and commercial settings [7–13]. These systems support applications such as processing financial market data, managing sensor network data collected from geographically dispersed sources and detecting network intrusion and other kinds of security violations in computer systems. Stream data sources typically produce large volumes of data at high and variable rates. Providing low latency, high throughput execution in the midst of dynamically changing data and network conditions as well as new, diverse processing requests is a challenge. One way to reduce traffic and improve performance of the DSPS is by dynamically placing stream processing tasks on the machines available in the stream processing network in order to maximally satisfy some objective function.

In this paper we present a biologically-inspired algorithm for dynamically placing tasks on a network of machines in a distributed stream processing system. Obtaining an optimal solution to the static task assignment problem, where the problem characteristics do not change, is computationally intractable [14]. The problem we address in this paper is even more difficult, due to the following additional requirements:

- **Dynamic and incremental**; We cannot assume that the entire flow graph and network description is known once and for all prior to deployment. The placement may need to react to changes, which may be (a) changes to the flow graph, such as a new query, or new consumer, (b) changes to the performance characteristics of the streams or operators, (c) changes to the availability or capacity of the machines or links. The system must react incrementally: that is, small changes to part of the system may induce small changes to the placement, rather than re-doing a single placement algorithm from scratch.

– **Decentralized**: We cannot assume a centralized server with global knowledge; instead we require that the placement execute in a distributed fashion on the same network of machines on which the stream processing system is executing.

Our biologically-inspired solution is grounded on techniques from ACO. ACO-based algorithms have several salient features that make them appealing for placement in distributed systems. ACO algorithms translate problems into finding optimal paths in graphs. Finding optimal paths is achieved through *stigmergy* [5], a method of indirect communication in a self-organizing emergent system. Stigmergy is achieved in ants through pheromone deposits. The concentration of pheromone guides ants towards appropriate routes. Simulating stigmergy in the context of ants in a computer system can be done in a completely decentralized manner using routing tables stored at every node. Successful global behavior is achieved by purely local decisions made at each node. The routing tables are updated by ant-like agents that carry very little state, and perform simple computations at nodes in order to update routes. Ant-like agents can be implemented online, in parallel with the functionality of the data processing system. As a result, the solution is naturally decentralized and dynamic.

Simulating stigmergy alone is not sufficient to achieve distributed task placement in stream processing systems using ants. Establishing an optimal path between the producers and consumers of a query requires ensuring that the path qualifies in terms of some quality of service metric (for instance the end-to-end latency between the producers and consumers of the query), as well as ensuring that the nodes in this path have sufficient capacity to process operators of the query without adversely affecting the performance of queries whose operators are already deployed on these nodes. The algorithm we present in this paper accomplishes both of these goals by introducing different "species" of ants and relying on a queueing model to estimate service time of tasks on a server. Placement in our algorithm is orchestrated by three different species of ants. *Routing ants* establish paths between the producers and consumers of a query by depositing pheromone in pheromone tables maintained at every node. *Scouting ants* estimate the cost of placing query operators on a set of machines in a path between a producer and a consumer of that query. *Enforcement ants* execute the placement of a query on a path. To the best of our knowledge, this work represents the first biologically-inspired algorithm for dynamic task placement in distributed stream processing systems.

The rest of this paper is organized as follows. After describing related work in Section 2, we provide complete details of our algorithm in Section 3. In section 4 we investigate our algorithm in simulation and end with conclusions.

2 Related Work

Operator placement has received significant attention in the distributed stream processing systems community. Existing literature consists of a wide variety of heuristic algorithms that range from static global optimization solutions to complete or partial decentralized solutions that perform local adjustments to placement dynamically during runtime.

Query Operator Placement in Distributed Stream Processing Systems In Flux, a dynamic load balanced strategy is developed in the context of continuous queries[15]. A centralized controller is responsible for collecting workload information and making load balancing decisions. Another approach computes placement by minimizing the average time, estimated by a queueing model, required for an event originating at the producer to reach its destined consumer, also using a centralized controller[16]. Operator placement has also been examined in the context of in-network stream query processing for sensor network environments with progressively increasing computational power network bandwidth up a hierarchy of nodes[14]. This approach provides theoretical analysis of a centralized placement algorithm that minimizes the total cost of computation as well as communication, but does not consider how the algorithm will respond to dynamic changes during runtime. A global optimization scheme for maximizing the weighted throughput of all queries in the system is proposed in [17]. Weights are provided as input and represent the importance or priority of a query operator. With the exception of[17], these placement schemes require a centralized controller to recompute the placement of the entire operator graph in order to respond to dynamic changes in the environment such as the introduction of new operators, changes in the network data rates, and changes in the availability of machines and network links. In [17], rather than recomputing an optimal placement in response to bursty data rates, a centralized controller jointly optimizes the input and output rates of operators, as well as their instantaneous processing rates. The global optimization scheme does need to be re-run however, when new operators need to be deployed, and when existing operators expire. Most recently a centralized placement algorithm has been proposed for a scheduler for *System S*, a distributed stream processing middleware, which balances the load on nodes and network traffic, and minimizes the inter-node traffic while respecting a host of constraints [18].

Decentralized algorithms have been proposed to minimize *network usage* and dynamically adjust placement in response to network changes during runtime[19, 20]. These algorithms focus on minimizing communication cost, and do not explicitly have a load-balancing strategy. One of these algorithms also considers reusing computation between overlapping queries [19]. It does not, however, compute the performance impact of reusing queries on existing queries. Reusing existing computation is important in certain stream processing applications where a majority of query processing requests are redundunt, such as in the financial services industry. An approach has been developed to address this that focuses exclusively on reusing component streams to satisfy new placement requests using a queueing-based quality-of-service impact projection algorithm [21]. This scheme does not outline how to compute an optimal placement when no existing computation can be reused.

Several decentralized algorithms only consider load management for computing an optimal placement [10, 22]. A data flow aware load selection strategy has been proposed in [23]. This approach aims to achieve lower communication cost by restricting the scattering of data flows, but does not assign placement by explicitly minimizing the end-to-end latency between the producers and consumers of queries. Furthermore, the load balancing scheme in this algorithm is based on partner selection which assigns a fixed number of load balancing candidate partners for each node, and load is moved individually for each machine between its partners. Another approach uses runtime monitoring information to adapt a decentralized placement algorithm that maximizes business utility which is defined as a function of the required bandwidth, available bandwidth and delay on a given edge of the network [24]. This approach proposes stream management middleware in which nodes self-organize into utility-aware clusters and requires cluster coordinators to maintain state for all nodes in a cluster. A component composition algorithm has also been proposed that dynamically composes quality-aware and resource-efficient stream processing applications from a system's currently available components while balancing the load [25]. Although this approach utilizes distributed composition probing, it requires global state.

Biologically-inspired Task Placement ACO algorithms have been applied to static task placement problems in which task and machine characteristics are fixed. In [26] the authors address the Quadratic Assignment Problem where each ant visits nodes and assigns a task to each node such that the product of the flows between activities is minimized by the distance between their locations. Tasks are assumed to be static and data rates are constant. Another variant is the Job-Shop Scheduling problem [27] where a job consists of an ordered sequence of operations. The problem is to assign the operations to time intervals in such a way that the maximum of the completion times of all operations is minimized and no two jobs are processed at the same time on the same machine. The static nature of the problems is a critical assumption that bolsters the success of these algorithms, and therefore they cannot simply be extended to solve dynamic task placement problems. In [6], an ant-colony optimization scheme is applied to dynamic traffic monitoring and routing. They outline how latency minimizing paths can be established between sources and destinations using ant colonies.

On the evolutionary front, considerable work has concentrated on applying genetic algorithms to static and dynamic task placement. None of these algorithms, however, are applicable to data stream processing systems where placement is concerned with tasks that are part of data flow graphs and execute on continuous streams of data.

3 Design and Algorithm

In this section we define our model, introduce necessary terminology, and present details of our ant-inspired task placement algorithm for distributed stream processing systems.

3.1 Stream Processing Model

We assume that every data source has knowledge of the flow graphs which specify the tasks executed on the data streams it produces. We also assume that stream processing tasks are asynchronous software components with multiple inputs, multiple outputs, and possible internal state. Tasks may execute asynchronously; messages sent between tasks are asynchronous. Task topology may change in response to dynamic requests to change the graph, but such changes are assumed to happen at rates much less frequent than the input streaming message rates. Servers can be actual machines, or processes or threads within a machine. At any point during execution, a server has some number of tasks assigned to it, representing a partition of the total dataflow graph. There is a single queue of messages waiting to be processed by the server. When the server is idle and the queue is non-empty, a message is dequeued, delivered to the appropriate task, and processed by that task, which may in turn generate internal messages for other downstream tasks in the same server. Processing of the message finishes when all such downstream tasks have finished executing. There may be one or more messages queued up for delivery to consumers or to downstream tasks in other servers; these messages are queued to links which are assumed to asynchronously transport these messages to consumers or to the queues in the appropriate servers. In this paper we do not address how to resolve problems such as lost messages that result from link or server failure. There are several well known techniques that address this [28].

Total latency depends upon the sum of server queueing delay, task processing time in each server, link queueing delay and link latency in each link, across the paths between producers and consumers.

In our approach, the streaming data messages and the tasks are augmented with ants (following our biological metaphor) and *cells*. Ants are implemented as special control messages distinct from data messages; cells are implemented as special tasks distinct from the tasks being placed. Each server contains one cell containing information about that server and its neighbors. Cells hold data, called *pheromones*, again following our metaphor; ants travel back and forth between cells carrying various information depending upon the kind of ant.

3.2 Approach Overview

Inspired by the success of Di Caro and Dorigo [6] in using ants to establish routes in telephone networks, we extend their work to accomplish task placement in a stream processing system. In particular we enhance their work to incorporate a queueing model as well as different species of ants. In addition to *routing* *ants*, we introduce *scouting ants*, and *enforcement ants*. These ants are responsible for placing each query. Placement is conducted through three stages. First, routing ants are dispatched by *leaders*, located at producers. Routing ants continuously travel between producers and destinations, depositing pheromones on their return trip in routing tables at every server that reflect different preference weights given to alternative next hops. Their destinations represent consumers of a query. Pheromone concentrations guide ants along best paths to their destinations. Once routing ants have established paths to the destination of a query, the leader dispatches scouting ants towards the same destination. Each scouting ant travels a particular path guided by pheromone concentrations towards its destination and computes the cost of a proposed hypothetical placement of the query along the path. Upon reaching its destination, it returns to the leader with its hypothetical placement report. In the third stage, after the leader has received enough reports from the scouts it had previously dispatched, it picks the best report, and dispatches an enforcement ant to execute the placement of the query outlined by the report. Once a query has been placed, the leader periodically dispatches routing and scouting ants for the query in order to ensure that its placement adapts to changes in the query tasks, as well as to changes in network conditions, data characteristics, and other queries being placed. If a more optimal placement of the query is found, then the leader picks the new placement, and dispatches enforcement ants to execute the new placement and discontinue the previous one. The concept of the pheromone vector and the role of routing ants are exactly as defined in the model by Di Caro and Dorigo [6]. The additional role of scouting ants and enforcement ants as well as the incorporation of the queueing model represent our novel contribution.

3.3 Pheromone Vector

There exists one cell per server which contains state relevant to an ant traversing this server. Each cell is aware of the set of destinations. Each cell C has a set of neighbors $N(C)$. For each destination D, a cell C maintains a pheromone vector [6] that maps each neighbor n into a probability $\Phi^C_{D,n}$ of choosing neighbor n as the next hop to travel from cell Cs server to destination D. Because these are probabilities, $\sum_{n \in N(C)} \Phi^C_{D,n} = 1$, for each destination D. Initially, when no information is known about one route versus another, the values of $\Phi^C_{D,n}$ may be set to be equiprobable (uniformly distributed). For each destination D, a cell C also maintains $\Gamma_C = \{\mu_{C \to D}, \sigma^2_{C \to D}\}$, where $\mu_{C \to D}$ is the mean and $\sigma^2_{C \to D}$ is the variance of the time to traverse both links and service queues on the best path from C to D.

Each query has one or more producers. Each producer site (or, if the producer is an external site, then at the server site which is the point of attachment of the producer, and which will act as a proxy for the producer for the purpose of this algorithm) contains a *leader*. Whenever a query is added, deleted, or changed, or whenever an unchanged query is requested to redo its placement, a leader at one producer is selected. If there is a single producer, the selected leader is the leader at that single producer. If there are multiple producers, the producer along the longest path to the consumer is selected. Thus the flow graph is clipped to a single linear chain of tasks, which we refer to as a *subquery*. Figure 1(b) shows the path selected by the ants for placement of the query in Figure 1(a) which has fan-in. Once the subquery is placed, the same algorithm is re-executed, one producer at a time, to place the tasks on paths from other producers that join in to a task on this path. Figure 1(d) shows the path selected by the ants for the placement of the query in Figure 1(c) which has fan-out. Once the subquery is placed, the same algorithm is re-executed, one consumer at a time, to place the tasks on paths that span out of the fan-out point.

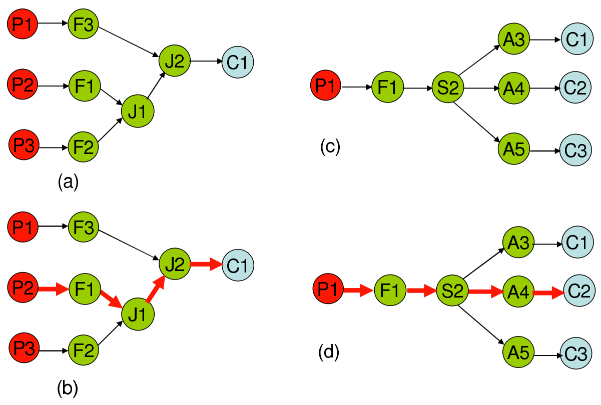


Fig. 1. (b) shows the path selected by ants for placement for the query in (a) which has fan-in. (d) shows the path selected by ants for the placement of the query in (c) which has fan-out. P and C stand for producers and consumers. F stands for the FILTER operator, J for JOIN, S for SPLIT, A for AGGREGATION.

3.4 Routing Ants: Forward Direction Seeking Paths

Intermittently, a leader residing at a producer of a query, releases a routing ant with destination D, the server ID of a consumer of the query, and source cell C_S. The number of ants and how frequently they are released are set as user defined parameters in our implementation. The ant carries on its back (i.e. the payload of a control message contains) the following information: its destination D, the path it has taken so far (the ID of each server hosting each cell it has passed through), and the delay it has experienced so far at each hop, passing through processing queues and link queues. Initially, only the source cell C_S is in the path. At each step, at a cell C that is not the destination D, the routing ant does the following:

1. It records on its back how long it has waited in the server queue at cell C.
2. It chooses a next hop towards D, by making a probabilistic choice of next hop neighbor, i, based upon the pheromone vector $\Phi^{C}_{D,i}$ at C, choosing among

the neighbors it did not already visit, or over all the neighbors in case all of them had been previously visited.
If a cycle is detected, that is, if an ant is forced to return to an already visited node, the cycle's nodes are removed from the ant's memory, and all information relating to them is destroyed. If the cycle lasted longer than the lifetime of the ant before entering the cycle, (that is if the cycle is greater than half the ant's age) the ant is destroyed.
3. It enqueues itself on the link towards the next hop neighbor, i, waiting to crawl through the link.
4. When it arrives at the next hop neighbor, i, it records on its back how long it has waited in queueing and propagation time passing through the link.
5. If it is not now at the destination D, it queues itself at the tail of the server queue of the next hop neighbor, i. Once it is dequeued, it repeats steps 1-5.
6. If it is now at the destination D, it turns around and begins its reverse journey back towards C_S.

3.5 Routing Ants: Reverse Direction Reinforcing Paths

Once a routing ant has reached destination D, it reverses direction and crawls backward towards the server C_S hosting its source cell. It knows how to reach its source, because it has stored on its back the path it had actually taken. On the reverse path, at each hop, it bypasses the server queues and goes directly to each cell C, to update its pheromone vector $\Phi^C_{D,n}$ and the estimates, Γ_C. After arriving at cell C from a cell $C-1$, the ant will update $\Phi^C_{D,C-1}$, which represents the probability of choosing cell $C-1$ as the next hop when attempting to reach D from C. The pheromone vector at cell C is updated by incrementing the probability $\Phi^C_{D,C-1}$ associated with neighbor cell $C-1$ and the destination D, and decreasing (by normalization) the probabilities $\Phi^C_{D,n}$ associated with other neighbor nodes $n, n \neq C-1$. The update procedure modifies the probabilities of the various paths, based on the *experience* the ant had recorded on its back when it chose the particular next hop neighbor on its forward path from C_S to D. While this experience can incorporate a variety of factors, in our implementation, we restrict it to $\frac{1}{T_{C_S \to D}}$, the inverse of the trip time, which can be computed using the information the ant loads on its back in the forward path to D, outlined in Section 3.4. Inverse trip time alone cannot be treated as an exact error measure, given its dependence on the load on the network. Therefore, the values stored in the model Γ_C are used to guide the adjustment of the trip times. Di Caro and Dorigo experiment [6] with a number of linear, quadratic and hyperbolic combinations of the trip time values and the estimates, Γ_C in order to create a reinforcement signal, $r \equiv r(\frac{1}{T_{C \to D}}, \Gamma_C), r \in [0,1]$. In our implementation, we define r as $\frac{1}{T_{C_S \to D}}$ and use Γ_C to decide when to update the pheromone vector. If the elapsed trip time of a sub-path is statistically *good* (i.e. it is less than $\mu + I(\mu, \sigma)$, where I is an estimate of the confidence interval for μ), then the time value is used to update the pheromone vector $\Phi^C_{D,n}$ and the estimates, Γ_C. On the other hand, trip times of sub-paths not deemed good, in the same statistical sense, are not used. The pheromone vector value $\Phi^{C}_{D,C-1}(t)$ at time t is increased by the reinforcement value at time $t+1$ as follows:

$$\Phi^{C}_{D,C-1}(t+1) = \Phi^{C}_{D,C-1}(t) + r.(1 - \Phi^{C}_{D,C-1}(t)) = \Phi^{C}_{D,C-1}(t).(1-r) + r \ . \quad (1)$$

Thus the probability is increased by a value proportional to the reinforcement received, and to the previous value of the node probability. Given the same reinforcement, small probability values are increased proportionally more than big probability values.

The probability $\Phi^{C}_{D,n}$ for all neighbor nodes $n \in N(C)$ where $n \neq C-1$ is decayed. This is essential to eliminate poor quality paths to D. These $n-1$ nodes receive a negative reinforcement by normalization. Normalization is necessary to ensure that the sum of probabilities for a given pheromone vector is 1.

$$\Phi^{C}_{D,n}(t+1) = \Phi^{C}_{D,n}(t).(1-r), n \neq C-1 \ . \quad (2)$$

After the routing ant has performed the reinforcement step on the cell at each hop back to the source, it dies. If a cell's pheromone vector for a destination is not updated beyond a given amount of time, the vector is destroyed. This ensures that a cell only maintains information that is relevant to the current set of queries deployed in the stream processing system, and furthermore prevents cell size from exploding arbitrarily.

3.6 Queueing Model

In this section we summarize the queueing-based flow performance model, presented in [16], which is utilized by ants in our algorithm to estimate the queueing delay experienced by a data packet in a server.

We define a *flow* as a path in a flow graph from a producer to a consumer that consists of an ordered sequence of tasks. Each server hosts a subset of the queries deployed in the stream processing network. This subset can consist of a number of logically unrelated segments of various flows, denoted as F. The service time of a flow is the sum of the service times of the tasks in the flow. Since some tasks are executed more than once if their ancestors in the execution sequence produce more than one event, batch sizes are incorporated in this calculation. For a given flow f, let θ_i be the set of tasks in the path from task t_i to the root task t_1, the entry task for events in this flow, with $\theta_1 = \{\}$. Let B_j represent the batch size of task t_j, task i's ancestor, such that $j < i$ and $j > 0$. The service time S_f of a flow $f \in F$ is the total amount of time a server is occupied due to an incoming event arriving at f, and this can be calculated as:

$$S_f = \sum_i S_i \prod_{j|t_j \in \theta_i} B_j \ . \quad (3)$$

The key insight driving the method in [16] is to first aggregate the input streams to a server into a single stream and simulate the behavior of all task flows through the server as one flow. Marginal metrics for individual flows can then be computed from the combined result. The aggregation/disaggregation approach proposed by Whitt [31] for servers with multiple incoming streams and multiple flows is appropriate for computing this. We begin by presenting the aggregation formulas applicable to our model.

The aggregate flow service rate, $\hat{\mu}$, the sum of the expected values of individual flow service rates, can be computed as:

$$\hat{\mu} = \frac{\hat{\lambda}}{\sum_f \frac{\lambda_f}{\mu_f}} \ . \tag{4}$$

where $\mu_f = \frac{1}{S_f}$ is the service rate of flow f and λ_f is its input rate, and the aggregate $\hat{\lambda}$ is the sum of the expected values of the individual flow input rates. The squared coefficient of variance for all the flow service times can be computed as:

$$c_s^2 = \frac{\hat{\mu}^2}{\hat{\lambda}} \left(\sum_f \frac{\lambda_f}{\mu_f^2} (c_{s_f}^2 + 1) \right) - 1 \ . \tag{5}$$

where $c_{s_f}^2 \equiv \frac{\sigma^2[S_f]}{E[S_f]^2}$ is the squared coefficient of the variance of flow f in which $E[S_f]$ is the flow's mean service time, and $\sigma^2[S_f]$ is the variance of the flow's service time. Assuming a general distribution for arrivals, Whitt's formula [31] can be used:

$$\hat{c_a^2} = (1 - w) + w \left(\sum_f c_{a_f}^2 \frac{\lambda_f}{\hat{\lambda}} \right) \ . \tag{6}$$

where

$$w = [1 + 4(1-\rho)^2(v-1)]^{-1} \ . \tag{7}$$

where

$$v = [\Sigma_f (\frac{\lambda_f}{\hat{\lambda}})^2]^{-1} \ . \tag{8}$$

where $c_{a_f}^2$ is the coefficient of variance for the flow f, and $\rho = \frac{\hat{\lambda}}{\hat{\mu}}$, is the load.

We can now use these to compute the expected queueing delay, Q_f, for a given flow in a server via a G/G/1 approximation due to Marchal[32]:

$$Q_f = \left(\frac{\rho}{1-\rho} \right) \left(\frac{\hat{c_a^2} + \hat{c_s^2}}{2} \right) \left(\frac{1}{\hat{\mu}} \right) \ . \tag{9}$$

We can now compute the expected latency L_f of a flow f through a server as the sum of its expected service time and the queueing delay:

$$L_f = Q_f + S_f \ . \tag{10}$$

We can model the delay experienced by a packet across a network link in the standard way employed in queueing theory. The link is modeled as a server, and a packet on the link experiences a queueing delay and a transmission delay. The queueing delay is a function of the link bandwidth and size of the messages crossing the link.

3.7 Scouting Ants: Hypothetical Placement

Once a threshold number of routing ants return, the selected leader dispatches multiple scouting ants. Each scouting ant carries information about the tasks in the subquery to be placed, as well as the ID of the server hosting C_S, the cell releasing the scout, and D, the ant's destination which is the consumer connected to the subquery. Each scouting ant will explore, in parallel with its team members, one hypothetical alternative for placing these tasks along a path from the given producer to the consumer. Exploration proceeds along cells selected on a hop-by-hop basis using the weighted probabilities in the pheromone vector in each cell. At each cell residing at a server, the scouting ant computes a hypothetical placement of tasks in the subquery. It uses a queueing model to estimate: (1) The given servers contribution to the latency of the hypothetically placed tasks, denoted as L_{new}; (2) the computational time of other tasks currently deployed on this server, denoted as L_{prev}, given the hypothetically placed tasks. Hypothetical placement calculations do not affect actual placement. The computational time of tasks currently deployed on the server has to be taken into account because it is affected and augmented by the additional stream volume introduced by a hypothetically placed task. In particular, when a new data stream is directed through a server, it affects the queueing delay experienced by data packets in all other data streams flowing through the server, and consequently affects the time it takes to service tasks on these other data streams. The ant greedily places tasks on a server provided that the sum of L_{new} and L_{prev} does not exceed a user-specified delay threshold, L_T:

$$L_{new} + L_{old} < L_T \ . \tag{11}$$

The queueing model summarized in the previous section is used to perform this estimate. If the delay threshold, L_T, is not specified by the user, then the ant ensures that the load on a server resulting from the combined existing and hypothetical flows does not become unacceptable. Specifically, the ant ensures that the load on a server, defined as $\rho = \frac{\hat{\lambda}}{\hat{\mu}}$ in section 3.6, where $\hat{\lambda}$ is the sum of the expected values of the individual stream input rates and $\hat{\mu}$, the sum of the expected values of individual stream service rates, is strictly less than a constant α, $\alpha \in [0, 1]$.

A parameter g, such that $g > 0$ and $g \in \mathbb{Z}$, represents the level of greediness of the scout, and controls the number of tasks the scout is willing to hypothetically place on the server, which is not necessarily the maximum number of tasks that can be placed on the server without violating the delay threshold, before moving on to the next server. The leader initializes scouts with randomly generated values for the parameters α and g.

At each cell, the scout records in its placement report (a) the ID of the server at which the cell resides, (b) which tasks it is placing on the server, and the sum of (c) the component of the latency of the hypothetically placed subquery at that server, and the impact on the latency of tasks in other subqueries deployed at that server. It then crawls to the next cell, using the pheromone vector exactly as the routing ants do. If it reaches the destination D without having been able to place all of its tasks, it declares failure and remains at the cell, C_D, in the destination server. If a number of failed scouts that traversed the same path with different levels of greediness, g, accumulate at C_D, one of them re-traces its steps to C_S and applies a negative reinforcement to the path in order to discourage future ants from pursuing the same path. If the scout succeeds in placing the tasks, it retraces its steps, and presents its scouting report to the producer. On its reverse path, the scout may conduct local load balancing of hypothetically placed tasks by estimating and comparing the service times of the same task on adjacent servers, and moving the task to the server on which placement is more efficient.

3.8 Enforcement Ants

If a scout succeeds in hypothetically placing all the tasks of the subquery, it returns to its dispatch leader, carrying a scouting report, consisting of the complete hypothetical placement, together with the latency statistics it recorded at each step. The leader waits a designated period of time for scouts to return. When a threshold number of scouts return the leader selects the best scouting report and dispatches enforcement ants to perform the placement outlined in this report. If a timeout occurs with too few scouts returning successfully, the leader may send out more scouts. Although, generally, the leader will wait for a threshold number of reports to return before making a decision about which placement to execute, if one of the scouts returns with an outstanding report, i.e. one with that has insignificant computational impact on the servers, the leader will proceed to execute this report without completing its wait for a threshold number of scouts to return. Before executing placement at each node, the enforcement ant will recalculate the placement cost of tasks that need to be placed at each server, and compare this against the statistics in the scout report. If the current cost of placement exceeds the cost stated in the scout's report by more than a threshold, the enforcement ant will return without executing placement, and the as a result the leader will dispatch more routing and scouting ants to discover other potential placements of the subquery. Thus, our placement strategy is most effective when local network conditions do not change significantly during the placement of a subquery.

Once the subquery has been placed, placement is recursively executed for other linear task chains in the flow graph. If a query has more than one producer, join points in the query are selected and the placement algorithm is recursively executed from each of these producers to this join point. Specifically, the ID of a server hosting a join point for a query is initialized as the destination in the routing, scouting and enforcement ants. These ants are released in parallel from each of the other producers of the query, and they are responsible for completing the placement of the subqueries that connect to the join point. For instance in Figure 1(b), the first subquery comprising of the chain of operators F1, J1 and J2 is placed. Then placement is re-executed from P1 to place F3 with J2 as the destination and from P3 to place F2 with J1 as the destination. If a query has fan-out, then multiple scouting and routing ants are dispatched from the producer to the point of fan-out. From this point the ants are dispatched in parallel to each consumer, in order to establish paths and place the remaining parts of the query between the point of fan-out and the other consumers. For instance in Figure 1(d), the first subquery comprising of the chain of operators F1, S2, and A4 is placed. Then placement is re-executed from S2 to place A3 with C1 as the destination, and from S2 to place A5 with C3 as the destination.

3.9 Updating Placement

Once a leader completes the placement of a query through enforcement ants, it periodically dispatches routing and scouting ants to seek more appropriate placements of the query in response to changing network conditions, such as changes in message rates, or changes in the network topology. The number of routing and scouting ants and the frequency with which they are periodically dispatched by a leader are initialized as user assigned parameters. The routing ants update routes to the consumers of the query. The scouting ants traverse the updated routes to determine new potential placements for the query. The leader periodically retrieves the current end-to-end service time for a query by sending an ant along the path on which the query is placed, and compares this with the service time of hypothetical placement reports of the query gathered by scouts. If the end-to-end latency of a hypothetical placement of a query is less than its most recently retrieved end-to-end latency (resulting from an existing deployment) by a factor β, such that $\beta > 0$ and $\beta \in \mathbb{R}_{\geq 0}$, then the leader dispatches enforcement ants to execute the new placement of the query, and discontinue the previous placement of the query. In the case where the query has one or more stateful tasks that must be moved as a result of the change in placement, there is a problem of conveying the state from the old location to the new location, or reconstructing the state at the new location. There are several well known techniques that address this [15, 29, 30], and we intend to examine this in future work.

Placement can also be explicitly updated by a user-initiated request in response to changes in performance or resource or flow graph characteristics. Users can initiate an updated query placement request accompanied with a request to terminate the deployment of the previous version of the query or request relevant producers to incrementally redo placement of one or more queries.

3.10 Task Reuse

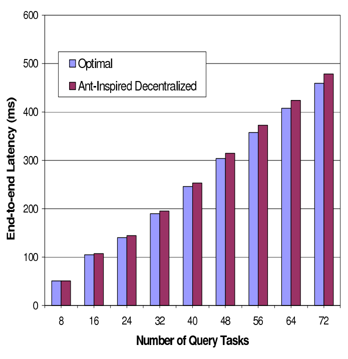
Operator reuse is incorporated into our model. When a scouting ant conducts hypothetical placement of a task on a server, it retrieves the currently hosted tasks on that server. If the scout finds a reusable task that produces the same result as the task to be placed, it reuses this task instead of hypothetically instantiating a new instance. When executing a placement order by the leader, enforcement ants first check for reusable tasks on each node instead of instantiating a new instance.

4 Experimental Evaluation

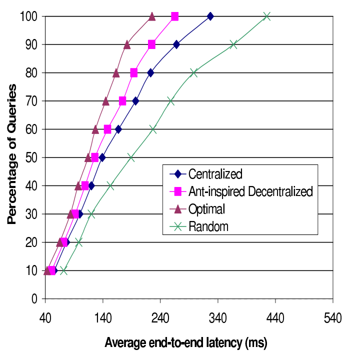
To evaluate the performance of our algorithm we implemented a discrete event simulator in Java. We randomly generated queries with both fan-in and fan-out. Each edge in a query graph is labelled with: (1) the *message input rate* in units of messages/millisecond, and (2) the *message size* in units of bytes/message. For each task, we define its *mean service time* in terms of a virtual work unit (VWU), which corresponds to a time unit (say, 1 second) on some standard machine such as an IBM ThinkPad T40. The VWU concept is mainly introduced to accommodate different processing capacities of the machines. Tasks in our queries include query operators such as SELECT, JOIN, PROJECT, AGGREGATION and SPLIT. Producers and consumers are assumed to be pinned to machines in the network. Although we randomly generated queries, we used meaningful values for the data on query edges and tasks that emulate workloads of data streams in the financial services industry.The network topology fed to the simulator was a transit-stub topology, generated by the GT-ITM internetwork topology generator. Nodes and links are assigned processing and communication capacities from discrete classes to simulate a heterogeneous system. The machine processing capacities are defined in units of VWU/millisecond such that if a T40 ThinkPad has processing capacity of 1, then a twice as fast machine would have a capacity of 2.

For comparison, we also implement three other common approaches: *optimal, random and centralized*. The optimal algorithm chooses the best possible placement with the lowest end-to-end latency based on an exhaustive search over all possible placements. The random algorithm selects a server for hosting each task at random and serves as a worst case comparison. We also compare against a centralized placement algorithm [16] whose goal is to produce an assignment of unpinned tasks to servers such that the expected average latency from producers to consumers over all paths is minimized. The centralized algorithm also utilizes a queueing model of the flow graph to determine how to compute the expected latencies due to the combination of delays for a given assignment. It employs a steepest descent search to find an approximate solution to the problem of minimizing the latencies. The algorithm accepts one flow graph as input which represents the concatenation of all queries that need to be placed. Placement of the entire flow graph has to be recomputed each time a new query needs to be placed, or there is a change in the data or network characteristics.

We evaluate our ant-inspired algorithm in terms of (1) the quality of its solution, (2) its adaptability and self-management capabilities, and (3) its scalability. Figure 2(a) compares the end-to-end latency of a flow graph placed by the optimal algorithm with our ant-inspired decentralized algorithm for an increasing number of query tasks. We observe that although the ant-inspired decentralized algorithm does not guarantee optimal results, the end-to-end latency of the query graph placed by it is not much worse than the end-to-end latency of the query graph placed by the optimal algorithm. Figure 2(b) compares the average end-to-end latency of all queries deployed on the network by the centralized, random and optimal algorithms with our ant-inspired decentralized algorithm. The total number of queries placed is 100. In this experiment 100 ants are released from each producer every 5 seconds. We observe that our ant-inspired decentralized algorithm consistently achieves better performance than other heuristic algorithms and similar performance as the optimal algorithm. While conducting this experiment, we also recorded the predicted running time of our distributed algorithm as output by our simulator and compared it against the running time of the centralized algorithm. We find that for queries compiled into flow graphs with more than 150 nodes, the centralized algorithm takes on the order of hours to run, where as the predicted running time of our algorithm is on the order of seconds.



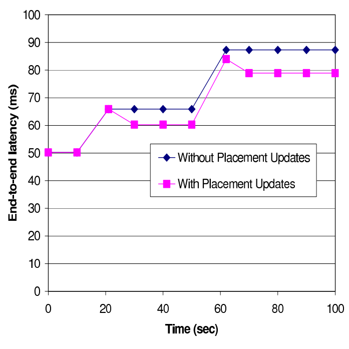
(a)



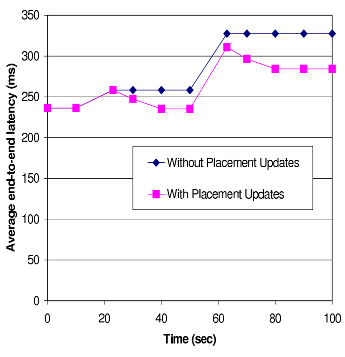
(b)

Fig. 2. Evaluation of the quality of our algorithm's placement solution. (a) Comparison of end-to-end latency achieved for increasing number of query tasks. (b) Comparison of end-to-end latency achieved for increasing number of queries.

Second, we evaluate the effectiveness of dynamically updating placement during runtime by our algorithm in response to changes in data characteristics. Figure 3(a) shows the variation in end-to-end latency for a 10-node query graph with and without self-optimization while network conditions are changing. During the 100 second simulation, the data rates at all producers of the query are increased by 10 ms at time 20 and then again by the same amount at time 60. We sample the end-to-end latency of the query every 10 seconds. The performance with self-optimization involving dynamic placement updates is clearly better than without self-optimization. Figure 3(b) shows the variation in average end-to-end latency for 200 concurrently executing queries with and without self-optimization in response to new queries being placed. 50 additional queries are placed at time 20 and at time 60. We observe that our decentralized and incremental algorithm generates a globally improved solution, as the average end-to-end latency over all queries, with self-optimization, decreases. We also observe that the change in average end-to-end latency consistently decreases and reaches a point where it stops decreasing, indicating that our algorithm does not continue to update placement indefinitely.



(a)



(b)

Fig. 3. Evaluation of our algorithm's adaptability and self-optimization capabilities. (a) End-to-end latency variation for a query with and without self-optimization when data rates are increased at time 20 and at time 60. (b) Average end-to-end latency variation of 200 concurrently executing queries with and without self-optimization when 50 additional queries are added at time 20 and at time 60.

Recall from section 3.9 that a decision to discontinue a query's placement and move it to a new placement is made when the end-to-end latency of a hypothetical placement of the query is less than its most recently retrieved end-to-end latency (resulting from an existing deployment) by a factor β, such that $\beta > 0$ and $\beta \in \mathbb{R}_{\geq 0}$. We calculate the loss in end-to-end latency incurred due to sub-optimal deployments of a 10-node query using different values of β in the presence of network perturbations (cross-traffic). The results are shown in Table 1. The loss is calculated as the integral over time of the difference between the maximum achievable end-to-end latency and the current end-to-end latency of the deployed flow-graph. The loss incurred is sufficiently low for a large number of values of β, and thus an appropriate value for β can be used to trade-off latency for a lower number of placement updates.

Table 1. Loss and number of placement updates for different values of β.

β	Number of Placement Updates	Latency-Loss (ms)
1.1	15	0
1.25	10	25.87
1.5	7	59.23
1.75	2	78.67
2.0	1	101.23

Finally, we evaluate the scalability of the our algorithm illustrated by Figure 4. We use different distributed stream processing systems with 200 to 600 nodes. As we add more nodes into the distributed stream processing system, the number of candidate servers for hosting tasks increases proportionally, so as to increase the capacity of the distributed stream processing system. We impose the same workload of 300 queries on those different distributed stream processing systems. Figure 4 shows the performance comparison results. We observe that our algorithm achieves similar scaling property as the optimal algorithm. For this experiment, we also recorded the message overhead added by our algorithm to the traffic in the network. We find that, on average, ant messages occupy 0.01% of the traffic on a link in the network of 200 nodes, and 0.022% of the traffic on a link in the network of 600 nodes.

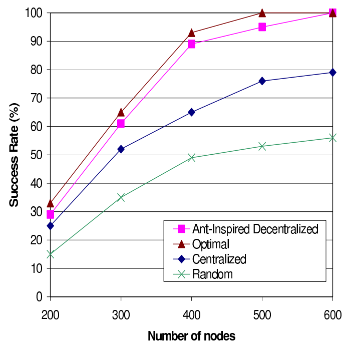


Fig. 4. Scalability of our algorithm: success rate of deploying 300 queries on distributed systems with different number of nodes.

5 Conclusion and Future Work

In this paper we have presented a biologically-inspired, distributed placement algorithm that reacts on-the-fly to placement requests of new flow graphs or to modifications of an already running stream processing flow graph, and dynamically adapts to changes in performance characteristics such as message rates or service times as well as to changes in processor availability or link performance during runtime. Our incremental algorithm is inspired by pheromone-based cooperation in ants and possesses many good properties that emerge as a result of this analogy such as completely decentralized control and no requirements for global state. Our simulation results show that our algorithm maintains scalable and effective self-management, while achieving high quality placement in terms of end-to-end latency. Although we choose to optimize placement for end-to-end latency, our model is generic enough to incorporate other metrics such as bandwidth or the product of bandwidth and latency. Instead of recording time at every hop along their forward routes, routing ants can record other metric related information from links and servers and use it to create and update pheromone entries on their return routes. Although we discussed and evaluated our algorithm in the context of tasks which are database query operators, it is applicable to the placement of any sequence of tasks on streams of data.

As future work we plan to develop a theoretical model of our algorithm and prove its correctness given concurrent placement of tasks by ants. We also intend to define some load placement primitives for each ant in order to prevent situations where placement continuously oscillates between two or more configurations. This can occur when data rates are particularly bursty. In addition to a queueing model, we would like to explore other methods to estimate the cost of hypothetical placement that are more suitable for real stream processing systems. We also intend to implement our algorithm on a real stream processing system.

References

1. Harmer, P.K., Williams, P.D., Gunsch, G.H., Lamont, G.B: An artificial immune system architecture for computer security applications. J. Evolutionary Computation, VOL. 23, NO. 6, 252–280 (2002).
2. Werner-Allen, G., Tewari, G., Patel, A., Welsh, M., and Nagpal, R.: Firefly-Inspired Sensor Network Synchronicity with Realistic Radio Effects. ACM Conference on Embedded Networked Sensor Systems, (2005).
3. Suzuki J. and Suda T.: A Middleware Platform for a Biologically Inspired Network Architecture Supporting Autonomous and Adaptive Applications. IEEE Journal On Selected Areas In Communications, VOL. 23, NO. 2, 249–260, (2005).
4. Lee, S-Y., Chang, H. S.:An ant system based multicasting in mobile ad hoc network. IEEE Congress on Evolutionary Computation, VOL. 2, 1583–1588, (2005)
5. Bonabeau, E., Dorigo, M., Theraulaz, G.: Swarm Intelligence. Oxford University Press, (1999)
6. Di Caro G. and Dorigo M.: AntNet: Distributed Stigmergetic Control for Communucation Networks. Journal of Artificial Intelligence Research 9 (1998): 317-365.
7. Exploratory Stream Processing Systems, `http://domino.research.ibm.com/comm/research_projects.nsf/pages/esps.index.html`
8. Financial Services: Real Time Data Processing with a Stream Processing Engine. White paper. `http://www.streambase.com/knowledgecenter.htm`
9. Abadi, D., et al.: The design of the borealis stream processing engine. In: Proceedings of CIDR, Asilomar, CA. (2005)
10. Cherniack, M. et al.: Scalable Distributed Stream Processing. In Conference on Innovative Data Systems Research, (2003).
11. Motwani, R., et al.: Query Processing, Resource Management, and Approximation in a Data Stream Management System, In Conference on Innovative Data Systems Research, (2003).
12. Chandrasekaran, S. et al.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World, In Conference on Innovative Data Systems Research, (2003).
13. Damani, O., Strom, R.: Smart Middleware and Light Ends for Simplifying Data Integration, In Conference on Information Reuse and Integration, (2006).
14. Srivastava, U., Mungala, K., Widom, J.: Operator Placement for In-Network Stream Query Processing. Proc. Principles of Distributed Systems, pp. 250-258, (2005).

15. Shah, M., Hellerstein, J., Chandrasekaran, S., Franklin, M.: Flux: An adaptive partitioning operator for continuous query systems. International Conference on Data Engineering, (2003).
16. Pandit, V., Strom, R., Buttner, G., and Ginis, R.:Performance Modeling and Placement of Transforms for Stateful Mediations, IBM Technical Report No. RI08002, 2004; http://www.domino.research.ibm.com/library/cyberdig.nsf/index.html
17. Amini, L., Jain, N., Sehgal, A., Silber, J., Verscheure, O.: Adaptive control of extreme-scale stream processing systems. International Conference on Data Engineering, (2006).
18. Wolf, J., et al. SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems. ACM Middleware, (2008).
19. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-aware operator placement for stream-processing systems. In: Proc. of 22nd ICDE, (2006)
20. Ahmad, Y., Cetintemel, U.: Network-aware query processing for stream-based applications. In: Proceedings of Very Large Data Bases (VLDB), (2004).
21. Repantis T., Gu, X., Kalogeraki, V.: Synergy: Sharing-aware component composition for distributed stream processing systems. ACM Middleware, pp. 322-341 (2006).
22. Balazinska, M., Balakrishnan, H., Stonebraker, M.: Contract-based load management in federated distributed systems. Symposium on Networked Systems Design and Implementation, (2004).
23. Zhou, Y., Ooi, B., Tan, K., Wu, J.: Efficient dynamic operator placement in a locally distributed continuous query system. International Conference on Cooperative Information Systems, (2006).
24. Kumar, V., Cooper, B., Schwan, K.: Distributed stream management using utility-driven self-adaptive middleware. International Conference on Autonomic Computing, (2005).
25. Gu, X., Yu, P., Nahrstedt, K.: Optimal component composition for scalable stream processing. In: 25th IEEE ICDCS, Columbus, OH (2005).
26. Maniezzo V., Colorni A., and M. Dorigo: The Ant System Applied to the Quadratic Assignment Problem. IEEE Transactions on Knowledge and Data Engineering 11(5), 769 (1998).
27. Colorni A., Dorigo M., Maniezzo V., Trubian M.: Ant System for Job-Shop Scheduling. JORBEL Belgian Journal of Operations Research, Statistics and Computer Science 34, 39–53 (1994).
28. Balazinska, M., Hwang, J.-H., Shah, M.:Fault-tolerance and high availability in data stream management systems. To appear in Encyclopedia of Database Systems.
29. Liu, B., Zhu, Y., Jbantova, M., Momberger, B., Rundensteiner, E.:A dynamically adaptive distributed system for processing complex continuous queries. In: Proceedings of Very Large Data Bases (VLDB), (2005).
30. Yang, Y., Kramer, J., Papadias, D., Seeger, B.: HybMig: A Hybrid Approach to Dynamic Plan Migration for Continuous Queries. IEEE Transactions on Knowledge and Data Engineering, Volume 19, Issue 3, Page(s):398–411, (2007).
31. Whitt, W.: The queueing network analyzer. Bell Systems Technical Journal, 66:2779-2813, (1983).
32. Marchal, W.: Some simpler bounds on the mean queueing time. Operations Research, 22:1083-1088, (1978).