Kinetically Stable Task Assignment for Networks of Microservers

Zoë Abrams Ho-Lin Chen
Leonidas Guibas
Computer Science Department
Stanford University
Stanford, CA 94305
zoea,holin@stanford.edu,
guibas@cs.stanford.edu

Jie Liu
Feng Zhao
Microsoft Research
One Microsoft Way
Redmond, WA 98052
liuj,zhao@microsoft.com

ABSTRACT

This paper studies task assignment in a network of resource constrained computing platforms (called *microservers*). A task is an abstraction of a computational agent or data that is hosted by the microservers. For example, in an object tracking scenario, a task represents a mobile tracking agent, such as a vehicle location update computation, that runs on microservers, which can receive sensor data pertaining to the object of interest. Due to object motion, the microservers that can observe a particular object change over time and there is overhead involved in migrating tasks among microservers. Furthermore, communication, processing, or memory constraints, allow a microserver to only serve a limited number of objects at the same time. Our overall goal is to assign tasks to microservers so as to minimize the number of migrations, and thus be kinetically stable, while guaranteeing that as many tasks as possible are monitored at all times. When the task trajectories are known in advance, we show that this problem is NP-complete (even over just two time steps), has an integrality gap of at least 2, and can be solved optimally in polynomial time if we allow tasks to be assigned fractionally. When only probabilistic information about future movement of the tasks is known, we propose two algorithms: a multi-commodity flow based algorithm and a maximum matching algorithm. We use simulations to compare the performance of these algorithms against the optimum task allocation strategy.

Categories and Subject Descriptors

A.m [**F.2**]: I.6

General Terms

Algorithms, Stochastic Processes, Matchings

1. INTRODUCTION

We consider the problem of task assignment in a network of resource-constrained computer nodes, which we call *microservers*. A task is an abstraction of a computational agent or a piece of data that is hosted by the microservers. Each microserver, due to communication, processing, or memory constraints, can only host a limited number of tasks at a time. Furthermore, due to physical or system dynamics, such as the motion of objects to be monitored or the motion of data consumers, the microservers that can host a particular task change over time. We also assume that tasks can be migrated among microservers for continuous execution, but this migration has certain costs, such as energy consumption and deterioration of quality of service; it is therefore something to be avoided whenever possible. At the microserver level this creates a dynamic resource allocation problem, in terms of deciding which microservers host which tasks.

Our overall goal is to assign tasks to microservers so as to minimize the number of migrations, and thus be *kinetically stable*, while guaranteeing that as many tasks as possible are hosted at all times.

As a concrete example, we motivate the problem by a vehicle tracking scenario in a parking garage. There are a set of camera sensors, each of which directly connects to a microserver. The cameras can pan and tilt to focus on specific areas in the garage. Each camera has a limited set of fields of view, and these fields of view may overlap, in the sense that multiple cameras can look at the same region. Generally, there will be multiple vehicles of interest driving in the garage that the cameras will track. The vehicle motions are somewhat predictable — there are road constraints in the garage as well as traffic patterns at various times of day that are known.

The microservers communicate through Ethernet among themselves and with a central server. A task is a video signal processing agent that can estimate the location of the vehicles it observes. We assume that due to video signal processing load, each microserver can only host a limited number of tracking tasks at a time.

When a vehicle performs a small motion, the tracking camera can pan or tilt to follow it. Over time, the vehicle may move out of one camera's field of view and must be tracked by another camera. Due to the design of the tracking algorithm, using the same camera to track a vehicle is easier than switching to another camera. Switching introduces the overhead of redetecting the vehicle from a different viewing angle, which may degrade the tracking performance. Tracking all vehicles may not be feasible. When conflicts on resource requests cannot be resolved, the video will be sent to central server for processing, but this should only be used as a last resort due to communication bandwidth limitations.

The goal is to minimize the number of task migrations, i.e. reduction of the vehicle redetection and task transfer overheads. Note

IPSN'06, April 19–21, 2006, Nashville, Tennessee, USA.

that some task migration is essential, in order for the system to adapt to the vehicle motion(s). Our focus is on avoiding unnecessary task migrations.

The migration of code and/or data among microservers in a sensor network is a generic problem that arises in several other contexts, beyond the tracking scenario discussed here. For example, in a setting where the users are mobile and operating in the same space as the sensors, we may want to migrate data of potential interest towards nodes near particular users, so that it is always accessible to them with low latency. Such migrating information caches may be of interest in many applications, from mobile telephony, to location-aware services, to search-and-rescue operations.

This paper defines and studies the kinetically stable task assignment (KSTA) problem. In section 2, we give a formal definition of the KSTA problem. In section 3, we analyze the deterministic KSTA problem, where the trajectory of objects are known. We show that the problem, even when the trajectories are deterministic and considering two steps, is NP-complete, has an integrality gap of at least 2, and can be solved optimally in polynomial time if we allow tasks to be assigned fractionally. In section 4, we study the problem for senarios where only probabilistic information about future trajectories is known, we propose two assignment algorithms: a multi-commodity flow based algorithm and a maximum matching algorithm. Section 5 compares the algorithms in a vehicle tracking scenario.

2. PROBLEM DEFINITION

We now formally define the *Kinetically Stable Task Assignment* problem. We assume that task allocations happen at discrete time steps indexed by integers $t = 1, 2, \ldots T$. N is the set of microservers and K the tasks. There is also an N length vector cap, where $cap(i)$ equals the capacity of microserver i. At each time step $t \in T$ there is a bipartite graph $B^t = (K, N, E^t)$ where $E^t = \{(u, v)|$task u can be hosted by microserver v at time $t\}$. A sequence of bipartite graphs is illustrated in Figure 1. We will consider scenarios where the edges of future bipartite graphs are known in advance and also where there is only probabilistic information about the future. We are looking for assignments of tasks to servers that are:

- *Maximum*: A solution to the problem will produce a maximum matching for every time step. Thus, if not all tasks can be covered at a given time step, the solution will cover as many tasks as possible, regardless of the number of migrations. For example, if there is the option that all tasks are covered and all tasks must migrate, this is preferable to covering all but one of the tasks without a single migration.
- *Feasible*: No microserver is assigned more tasks than it can process (i.e. microserver i is assigned no more than $cap(i)$ tasks at each time step).
- *Stable*: A feasible assignment has to be maintained as the tasks move around. Under the two constraints above, the primary optimality criterion is the *stability* of the assignment, which is measured by the sum over all tasks of the number of times a task has to be switched to a new server during a period of task motions. Precisely, the number of migrations at time step t is the number of edges in the matching for B^t that do not exist in the matching for B^{t-1}. The goal is to produce a matching for all the bipartite graphs such that the total number of migrations from time zero until time T is minimized.

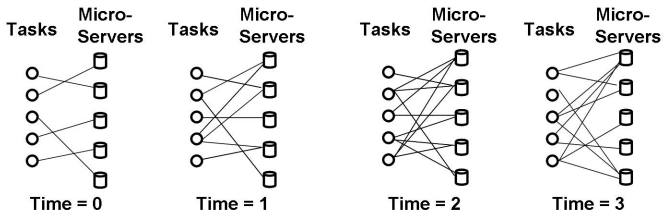

Figure 1: A sequence of bipartite graphs.

We will further assume a setting where:

- Time synchronization is available for microservers.
- All data traffic is quickly routed through microservers; in particular, we have a (multi-hop, if necessary) routing protocol for getting data from the actual sensor nodes to nearby microservers.
- Source interference can be resolved at the microserver or high-end server level, so that source separation is not an issue.
- Higher-level processes can make predictions (though with uncertainty) about the motion or evolution of the phenomena of interest.
- Energy and communication are not explicitly considered. Migrations incur a fixed cost (as opposed to factoring in communication which might vary based on distance, etc.).

If we set microserver capacities and the number of time steps equal to 1 and look at the static setting only, this is the classical and very well studied *assignment* or *bipartite matching* problem. Distributed algorithms exist for its solution, such as the market-based auction algorithm [4]. As we will discuss later, our heuristics take advantage of this previous research.

3. DETERMINISTIC KINETICALLY STABLE TASK ASSIGNMENT

In the Deterministic Kinetically Stable Task Assignment problem, the future trajectories of tasks are fully known. The bipartite graphs that will arise in future time steps are known with certainty, as is the bipartite graph in the current time step (Time = 1) and the assignment from the previous step (Time = 0).

3.1 NP-Completeness Reduction

We now show that finding an integral solution to the Deterministic Kinetically Stable Task Assignment problem, even when there are just two time steps, is NP-complete, via a reduction from MAX SET COVER.

THEOREM 1. *Given two bipartite graphs $G = (V, E)$ and $H = (V, E')$, finding a maximum matching for each graph, so that the number of edges in common between matchings is maximized, is NP-complete.*

PROOF. The reduction is from MAX SET COVER. An instance of MAX SET COVER is as follows: Given a set of elements $S = A_1, A_2, \ldots, A_n$ and m subsets of these elements $C_1, C_2, \ldots,$

C_m, the objective is to find k subsets that cover as many elements in S as possible. Given such an instance, we construct the following Integral Deterministic Kinetically Stable Task Assignment problem:
List of microserver and task nodes:

1. For every element $A_j \in S$ and every subset C_i, if $A_j \in C_i$, there is one microserver node $S_{i,j}$ and one task node $K_{i,j}$.

2. There are m 'link' microserver nodes $S_{link,1}$, $S_{link,2}$, ..., $S_{link,m}$ and m 'link' task nodes $K_{link,1}$, $K_{link,2}$, ..., $K_{link,m}$.

3. There are k pairs of 'set' microserver and task nodes $S_{set,1}$, ..., $S_{set,k}$, $K_{set,1}$, ..., $K_{set,k}$.

4. If an element A_j appears in t different subsets, then there are $t-1$ pairs of 'element' microserver and task nodes $S_{elm,j,1}$, $S_{elm,j,2}$, ... $S_{elm,j,t-1}$, $K_{elm,j,1}$, $K_{elm,j,2}$, ..., $K_{elm,j,t-1}$.

Edges for Time Step 1*:* As shown in figure 2,

1. There is an edge between every 'set' microserver node $S_{set,i}$ and every 'link' task node $K_{link,j}$.

2. There is an edge between every 'link' microserver node $S_{link,i}$ and every 'set' task node $K_{set,j}$.

3. There is an edge between $S_{i,j}$ and $K_{i,j}$ for every $A_j \in C_i$.

4. For every subset C_i, if the subset C_i contains elements A_x, A_y, A_z, then we add edges between $K_{link,i}$ and $S_{i,x}$, between $K_{i,x}$ and $S_{i,y}$, between $K_{i,y}$ and $S_{i,z}$ and between $K_{i,z}$ and $S_{link,i}$, as shown in figure 2.

5. There is an edge between $S_{elm,i,j}$ and $K_{elm,i,j}$ for every i, j.

Edges for Time Step 2*:* As shown in figure 2,

1. There is an edge between $S_{i,j}$ and $K_{i,j}$ for every $A_j \in C_i$.

2. There is an edge between $S_{elm,j,x}$ and $K_{i,j}$ for every i, j, x.

3. There is an edge between $K_{elm,j,x}$ and $S_{i,j}$ for every i, j, x.

4. There is an edge between $S_{link,i}$ and $K_{link,i}$ for every i.

5. There is an edge between $S_{set,i}$ and $K_{set,i}$ for every i.

In the above construction, the only common edges between time 1 and time 2 are the edges between $S_{i,j}$ and $K_{i,j}$ for every i, j. In a perfect matching for time = 1, the 'set' microserver nodes will be paired with exactly k 'link' task nodes. Hence there are exactly k values i_1, i_2, ..., i_k such that the edge between $S_{i_x,j}$ and $K_{i_x,j}$ is used. At time = 2, we can get a perfect matching between microserver nodes and task nodes, if and only if for every element A_j, there is exactly one edge between $S_{i,j}$ and $K_{i,j}$ being used in the matching. So we can have M edges commonly used in the two perfect matchings if and only if we can find k subsets $C_{i_1}, C_{i_2}, \ldots, C_{i_k}$ such that these subsets cover M elements in the set S.

□

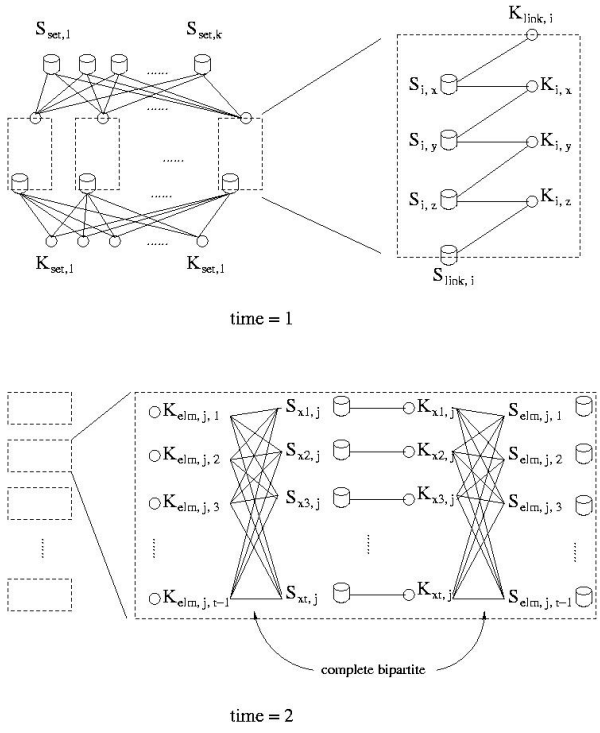

Figure 2: The construction of the NP-Completeness proof.

3.2 Multicommodity Flow Linear Program Formulation

The above negative result leads us, for now, to relax the requirement that tasks be assigned integrally. In some settings, splitting a task across multiple microservers may indeed be reasonable. Under this assumption, we can write the problem as a linear program using a multicommodity flow formulation. In this formulation, there is a node v_{it} for every $i \in N, t \in T$. All commodities originate from the same source node s and the destination for all commodities is sink node $\bar{t}$. There are edges from node s to nodes v_{i1} for all $i \in N$, from v_{it} to $v_{jt'}$ only if time t' follows time t, and edges from nodes $v_{iT}, \forall i \in N$ to the sink $\bar{t}$. The linear program uses non-negative variables x^k_{ijt} to indicate whether or not task k is moved from microserver i to j at time step t. With a slight abuse of notation, if the time is zero (or T), then the source (or sink), is considered the microserver for that time slot. The cost of migrating task k from microserver $i \in N$ at time step t to microserver $j \in N$ at time step $t+1$ is denoted c^k_{ijt}. We set c^k_{ijt} to 0 if $i = j$ or if either i or j is the source or sink, and the cost is 1 otherwise. An emergency node is used if a complete matching does not exist. The cost of using the emergency node is set to be prohibitively expensive, ensuring a maximum matching is chosen at every time step. We assume that infeasible indicator variables are zero (i.e. if task k cannot be monitored by microserver j at item t, then $\forall i \in N$, $x^k_{ijt} = 0$ and $x^k_{ji(t+1)} = 0$). The linear program can be written:

$$min \sum_{kijt} c^k_{ijt} x^k_{ijt} \quad (1)$$

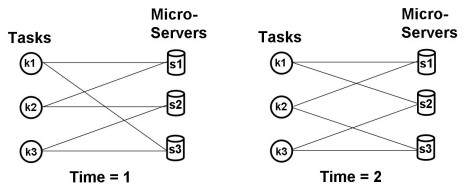

Figure 3: Illustration of an integrality gap of 2.

$$\forall i \in N, t \in T \qquad \sum_{kj} x^k_{ijt} \leq cap(i)$$

$$\forall i \in N, i \neq s, \bar{t}, t \in T, k \in K \qquad \sum_{j} x^k_{ijt} - \sum_{j} x^k_{ji(t-1)} = 0$$

$$\text{for source node } s, \forall k \qquad \sum_{i} x^k_{si0} = 1$$

3.3 Integrality

The linear programming formulation given above can only be used if the solution can be fractional (i.e. the monitoring of a task can be split so that microservers monitor fractional portions of the tasks). Though this may be acceptable for specific applications, if we require an integral solution, there is the possibility that the solution generated by the LP will be fractional and therefore infeasible. We now show that, in fact, there exist scenarios where the optimum fractional solution is twice as good as any optimum integral solution.

We illustrate this integrality gap of 2 in the example of Figure 3. In the first time step, without loss of generality, let us assume we assign task $k1$ to microserver $s1$, task $k2$ to the microserver $s2$, and task $k3$ to microserver $s3$. In the second time step, if we require an integral solution, either task $k1$ stays assigned to microserver $s1$ and $k2$ and $k3$ switch, or task $k3$ stays assigned to microserver $s3$ and tasks $k1$ and $k2$ switch. Either way, two tasks must migrate. If we allow fractional assignments though, then each microserver can monitor half of both the tasks it covers initially. In the second time step, half of task $k1$ migrates from microserver $s3$ to microserver $s2$ and half of task $k2$ migrates from microserver $s2$ to microserver $s3$, for an optimum total of 1 migration — twice as good as when the solution was required to be integral.

4. PROBABILISTIC KINETICALLY STABLE TASK ASSIGNMENT

In the Probabilistic Kinetically Stable Task Assignment problem, the goal is to use the previous assignment and *probabilistic* information about future movement of the tasks to minimize the expected number of times tasks migrate between microservers. As opposed to the deterministic case, there is only probabilistic information about the future microservers that will be available to cover a specific task.

4.1 Motivating Examples

We now give examples that will motivate the need for using probabilistic information about task trajectories to determine the current assignment. There are three types of scenarios in which future probabilistic information is useful. First, there are situations where the number of matchings available for the algorithm to choose from decreases over time. Second, situations arise when there is contention for a microserver, and this contention forces a choice that will be detrimental in the future. Finally, there are situations where future information can allow us to anticipate which microserver will be most useful later on. For the first two scenarios, we give an extreme example to emphasize the drastic implications of future knowledge, and a more realistic example to show that these scenarios may actually occur often in practice.

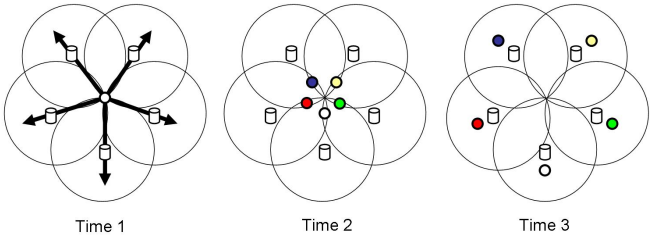

Figure 4: As the tasks move outward, lack of foreknowledge leads to additional migrations.

- **Dwindling Options.** Consider a complete (K, N) bipartite graph with $K = N$. In the first time step, we choose an arbitrary matching. In the next time step, all edges in the bipartite graph remain except the edges from the matching chosen in the previous time step. We continue this process for N time steps, and at each time step all tasks are required to migrate. We know that a perfect matching will exist at every time step from [8], leaving K^2 total migrations. In contrast, had the edges in all future time steps been known in advance, then the matching remaining in the last time step could have been chosen first, resulting in zero total migrations.

  Although compelling because of the dramatic advantage obtained due to future knowledge, the previous example is not particularly realistic. Figure 4 depicts a more realistic scenario. The trajectories of the tasks are illustrated with bold arrows. In the first time step, all tasks are situated at the center. Every microserver can monitor every task and, therefore, any assignment can be chosen from amongst all 5! possibilities. In the second time step, there are 13 possible matchings since every task can now only be monitored by 3 microservers. Finally, at Time 3, there is only one single possible matching of tasks to microservers. An algorithm that has no knowledge about the task trajectories will choose a matching uniformly at random from all 5! possibilities in Time 1. At Time 2, the algorithm will choose a matching from all 13 possibilities that avoids as many migrations as possible. For each of the 13 possibilities, the expected number of migrations is at least 2 since the probability that a task is no longer capable of being monitored by a microserver is $\frac{2}{5}$, and matching requirements only decrease the likelihood that a task can continue to be monitored by the microserver from Time 1. Furthermore, each of the 13 possibilities is equally likely to be chosen. At Time 3, averaging over all 13 choices in the previous time step, the expected number of migrations is $\frac{40}{13}$. The above example can be duplicated an arbitrary number of times, resulting in $\frac{66K}{13}$ expected migrations over 2 time steps compared with zero optimum migrations.

- **Contention.** If many tasks can be assigned to a microserver, but only a few tasks will benefit from that assignment in the future, then the multitude of contenders for service creates a situation where future knowledge is beneficial. To illustrate this, we consider n tasks $k_1, k_2, ...k_n$, $2n$ microservers

$s_1, s_2, ..., s_{2n}$, and n time steps. $\forall i$ such that $2 \leq i \leq n$, task k_i can be assigned to microserver s_i in all time steps. Microserver s_1 can monitor all tasks in all time steps. In addition, task k_1 can be monitored by microserver s_{n+t} at time t. The optimum solution that knows the trajectories of the tasks will assign every task k_i to microserver s_i and incur no migrations. If the future trajectories of the tasks are not known in advance, then initially there are $n+1$ possible matchings (when k_1 is assigned to s_1 there is 1 possible matching and when k_1 is assigned to s_{n+1}, sensor s_1 can be unassigned or assigned any of the other $n-1$ tasks). There is no way to discern which matching to choose, and if s_1 monitors a task other than k_1 initially, then at each consecutive step the matching that requires the least migrations is to migrate only task k_1 resulting in a total of n migrations (one at each time step).

A more common example of where contention pushes the choice of the assignment in the wrong direction is given in Figure 5. Each microserver monitors the rectangle of which it is the center. Initially, the task $k1$ is monitored by the microserver $s5$ and task $k2$ is monitored by the microserver $s1$. Both tasks move out of range and must be reassigned ofter two time steps. The movement is indicated with the black arrows. In the future, the tasks will continue to move in the same direction and the optimum assigns the task $k1$ to microserver $s2$ and task $k2$ to microserver $s3$. However, contention for the microservers $s2$ and $s3$ push the task $k1$ assignment in the wrong direction. Out of the four possible matchings, two assign task $k1$ to microserver $s4$, so this is the most probable assignment, resulting in an increase in expected migrations compared with the optimum.

- **Anticipation** An even more common example is a car moving along a path at an even pace with microservers spaced evenly along the path. Precisely, there is a line of microservers, $S_1, S_2, ...S_k$, and if the task can be monitored by microservers S_i and S_{i+1} at time t, then it can be monitored by microservers S_{i+1} and S_{i+2} at time $t+1$. If the algorithm knows the trajectory in advance, it will migrate once every two time steps. If the algorithm has no advance knowledge about the direction the car will follow, it must make a random choice between the two options available every time the task moves out of sensing range. The expected number of migrations at time t, $E[t]$, is the probability that the microserver monitoring at time $t-1$ was microserver S_{t-1}. Let h^t be the number of events ending with microserver S_t monitoring the task at time t. Then h^t can be defined by the recursive formula $h^t = h^{t-1} + h^{t-2}$, since monitoring can be transfered from previous time $t-1$ or $t-2$. The probability that microserver S_t monitors the task at time t is $\frac{h^t}{h^t + h^{t-1}}$, which in the limit, as t approaches ∞, approaches $\frac{\phi}{\phi+1} \approx .618$, where ϕ is the golden ratio. The increase in migrations is approximately 20%.

4.2 Multicommodity Flow (MCF) Algorithm

Given current time t, we find a matching for the bipartite graph of the current time step $B^t = (K, N, E^t)$ *and* the bipartite graphs for all future time steps. The value of the matching is the time-discounted[1] number of migrations within a time window plus the time-discounted likelihood that the matching will be possible. Let $pr(k, j, t)$ be the probability that task k can be assigned to microserver j at time t and let $0 < \beta < 1$ be some time discounting factor. Let C be a predefined constant. Precisely, the objective function minimized is:

$$c^k_{ijt} = \begin{cases} 1000 * K * T & \text{if } j \text{ is emergency node} \\ (-log(pr(k,j,t) - \epsilon))(\beta^t) & \text{if } j = i \\ (C + -log(pr(k,j,t) - \epsilon))(\beta^t) & \text{otherwise} \end{cases}$$

This objective function can be explained with a generative model. Consider each task as an agent working on its own behalf with no concern for the constraints of the overall system (i.e. the microserver capacities). Then the probability that a task will choose a given trajectory is the product of probabilities that the choice was available and that it was the best option at that time. Taking the log we have the sum of migrations that occur plus the sum of the log of probabilities along that edge. We can think of the objective function as trying to let as many tasks choose as they please subject to the capacities of the microservers.

An appealing property of the multicommodity flow algorithm is that it approaches the optimum as certainty about the trajectories increases. It also plans for future assignment constraints to avoid additional migrations due to the maximum matching requirement. It solves a linear program with N^2WK variables and $\Theta(KNW)$ constraints. The solution can be computed quickly using commercial linear programming software such as CPLEX.

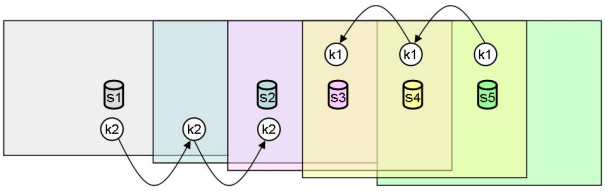

Figure 5: Additional migrations due to contention.

4.3 Matching Algorithm

Given current time t, we find a maximum matching of minimum cost for the bipartite graph $B^t = (K, N, E^t)$ where the cost of an edge k, n in this matching is

$$c^k_n = \begin{cases} 1000 * K * T & \text{if } n \text{ is the emergency node} \\ -\sum_{t=1}^{T} (\beta^t) pr(k, n, t) & \text{otherwise} \end{cases}$$

The matching algorithm is an attractive algorithm because of its simplicity and because it is fast. It can be solved using standard centralized algorithms with running time $\Theta(max(K, N)^3 + KNW)$ (recall that W is the size of the look-ahead window). Also, there are several possible distributed implementations [4, 2, 6, 7] that can be adapted to our setting.

4.4 Examples for Comparison

The MCF Algorithm Outperforms the Matching Algorithm

The multicommodity flow algorithm performs better than the matching algorithm when the choice of edges in the future is heavily dependent on the existence of a matching that contains those edges. This concept is illustrated with an example in Figure 6. Both algorithms have complete knowledge of the next time step. There is only one matching in the second time step and it con-

[1]By time discounted, we mean that migrations that are in the near future are more costly that those in the distant future.

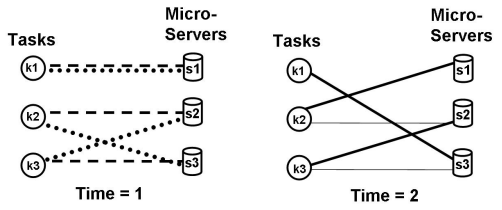

Figure 6: Example where the multicommodity flow algorithm outperforms the matching algorithm.

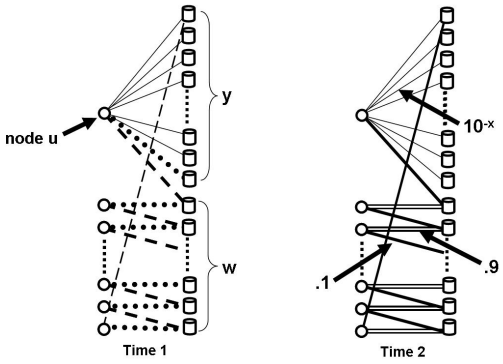

Figure 7: Example where the matching algorithm outperforms the multicommodity flow algorithm.

tains bold edges $(k1, s3)$, $(k2, s1)$, and $(k3, s2)$. The MCF algorithm will choose the dotted matching $(k1, s1)(k2, s3)(k3, s2)$ because it is the matching that requires the fewest migrations overall. The matching algorithm will choose the dashed matching $(k1, s1)$, $(k2, s2)$,$(k3, s3)$ because 2 edges in this set are also present in the next time step and therefore the cost of the matching is -2. Alternatively, the dotted matching $(k1, s1)(k2, s3)(k3, s2)$ has a more expensive cost of -1.

Matching Algorithm Outperforms the MCF Algorithm

The multicommodity flow algorithm is especially vulnerable in situations where the existence of a *particular* matching is not likely, but there is a high probability that *some* matching with a particular set of edges will exist. Figure 7 illustrates this vulnerability. In the second time step, the horizontal double lines exist with probability .9, the bold edges exist with probability .1, and all other edges exist with probability 10^{-x}. The multicommodity flow algorithm evaluates the matching in time step 1 made up of dashed edges as having cost $(w + 1)(-log\frac{1}{10}) = w + 1$. The matching consisting of dotted edges has cost $w(-log\frac{9}{10}) - log10^{-x} = .046w + x$. For sufficiently large x, the matching with dashed edges is less expensive and the algorithm will choose this matching. However, the matching made up of dotted edges will have fewer migrations in expectation because each of the horizontal edges in time step 2 is highly likely, and the probability that *some* edge connects node u to one of the first y microservers is $(1 - 10^{-x})^y$, which could be arbitrarily close to 1 for large y. In contrast, the matching algorithm will prefer the dotted matching at cost $\frac{-9}{10}w - 1^{-x}$, which is smaller than $\frac{-1}{10}(w + 1)$.

5. SIMULATIONS

5.1 Problem Instances

A *problem instance* consists of

- **Problem Parameters:** Problem parameters include a set of locations P, a set of tasks K, and a set of microservers N. The number of time steps that tasks will have to be monitored in the future is denoted T. The time window W is the number of time steps in the future that the algorithm considers when determining its matching in the current time step.

- **Markov Chains:** We use a Markov chain to represent the probabilistic information known about task movement. There is a state for each discrete location where a task can be placed, and a transition probability from state s to s' represents the probability that the task will move to the location that corresponds with state s' in the next time step, given that it is located at the location that corresponds to states s in the current time step. There is a separate transition matrix defined for each task, but the set of states over which the transition matrix is defined is consistent between tasks. There is a P by P transition matrix M for every task $k \in K$ such that $M_k(i, j)$ is the probability that task k moves to location i in the next time step if it is at location j in the current time step.

- **Microserver Coverage:** Each microserver covers some subset of locations, and has the ability to monitor any task at a location it covers. The microserver coverage is represented with a P by N matrix C such that $C(i, j) = 1$ if microserver j covers location i, and equals zero otherwise.

- **Microserver Capacities:** Each microserver has a capacity limiting the number of tasks it can monitor in a single time step. The capacity values are stored in an N length vector cap, where $cap(i)$ equals the capacity of microserver i.

- **Initial Assignment:** A K by N matrix A where $A(i, j) = 1$ iff microserver j monitors task i initially.

- **Trajectories:** The actual trajectory taken by each task is held in a P by K by T matrix J where $J(i, j, t) = 1$ iff task j is located at location i at time t. In Figure 8, the trajectories of 4 tasks (indicated by the colored arrows) over 4 time steps (time starts at the tail of the arrow and ends at the head) is shown.

An algorithm for the task assignment problem will determine an assignment of tasks to microservers at each of several time steps. It is given time invariant input: M, cap, C, and T. These inputs do not change throughout the course of the simulation. In addition, there are several inputs to the algorithm that will change during the simulation including the initial assignment. Information about the future trajectories of the tasks is hidden from the algorithm. The algorithm can only use the Markov Chains to predict trajectories.

5.2 Baseline Settings

Unless otherwise indicated, the settings for our simulations are shown in the table below. In Figure 9 we see the Markov chain used by the baseline settings and also the microservers along with the locations they cover.

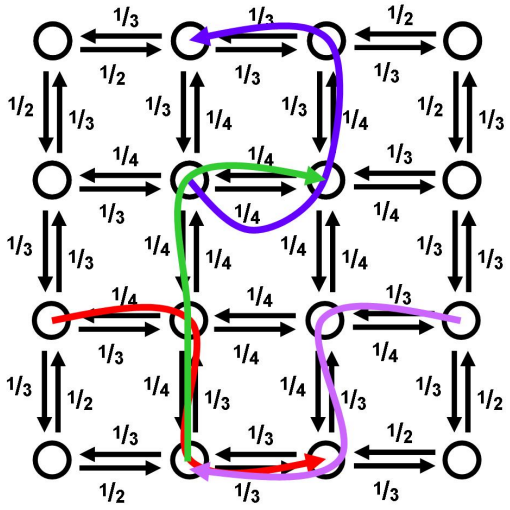

Figure 8: Task trajectory through a Markov Chain.

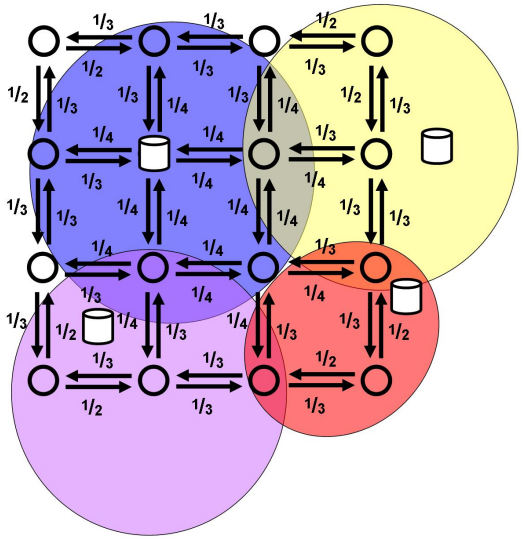

Figure 9: Locations, transition probabilities, and microservers.

Component	Baseline Setting
Total Time T	16
Time Window W	4
Locations P	16
Time Discount β	.8
Cost Constant C	1
Microservers N	4
Capacities *cap*	Each microserver has capacity 1.
Tasks K	4
Initial Assignment	An arbitrary maximum matching of tasks to servers chosen at the start of the simulation.
Microserver Coverage	There is a microserver at every location on the grid with even row and column indices. A microserver covers its location, its neighbors, and if it exists, the location at its right and bottom diagonal. There is also an emergency microserver covering all locations.
Markov Chain Topology	Locations are arranged in a grid and every task has the same Markov chain topology. Every location has either 4 neighbors (if it is internal), 3 neighbors (if it is along an edge), or 2 neighbors (if it is on a corner). Every location is a neighbor of itself (i.e. all locations have self loops).
Markov Chain Probabilities	The *skew*, $\gamma = .5$, signifies that one neighbor is transitioned to with probability $\gamma + \frac{(1-\gamma)}{\text{neighbors}}$ and all other are transitioned to with probability $\frac{(1-\gamma)}{\text{neighbors}}$. The chosen neighbor is selected uniformly at random from all neighbors and is the same for all tasks.

5.3 Results

In the simulations, although the multicommodity flow algorithm did produce some fractional solutions, fractional solutions were rare. Less than $\frac{1}{500}$ of the solutions were fractional and of these, all were half integral. Since fractional solutions were rare, the simulations analyzed in this section are based solely on integral solutions.

An algorithm's performance is measured by the number of migrations it performs. We use g_{OPT}, g_{Match}, and g_{MCF} to denote the number of migrations in each respective algorithm. The percentage increase in migrations ($\frac{g_{alg}-g_{OPT}}{g_{OPT}}$) compares each algorithm against the optimum. The number of migrations is most meaningful relative to $\mathcal{C}$, the number of tasks that *can* be covered (i.e. the sum, over all time steps, of the maximum matchings in the bipartite graphs). The weighted number of migrations is the migrations divided by $\mathcal{C}$. All data is averaged over 10 problem instances. The random variables used to create the problem instance will be chosen anew for each trial. We observe changes in performance as the value of certain components in the baseline are varied.

Variation in Quantity of Tasks

There is a large percentage increase in the number of migrations as the number of tasks in the system is varied from 2 to 18 by increments of 4 as shown in Figure 10. The average percentage increase in migrations compared with the optimum increases from .06 to .5 for the multicommodity flow algorithm and from .07 to .47 for the matching algorithm. This may be attributed in part to the decrease in the overall number of migrations. However, this does not account for the difference entirely, since the number of additional migrations in the MCF and Matching algorithms increases, even as the overall migrations decreases. The decrease in performance is attributed to the additional option that are available to the optimum. The more tasks available, the more likely it is that there will be tasks (known only by the optimum) that require less migrations. As there are more options, having full knowledge of these options becomes more valuable.

Observe in Figure 11 that there is an increase in the overall number of switches as the number of tasks increases from 1 to 8. Despite there being the same tasks with the same trajectories present in the variation with 8 tasks as the variation with only 1 task, the algorithm cannot simply apply the same solution as was used in the variation with 1 task since it is *required* to monitor as many tasks as possible. The total number of tasks that must be monitored at each time step increases as the maximum matching in each time step increases, pushing up the number of migrations.

Skew Variation

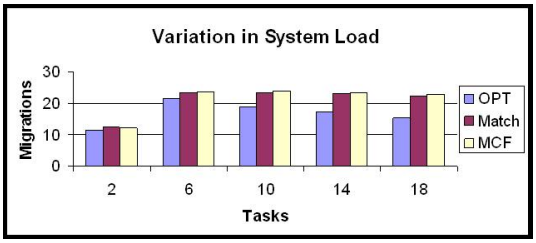

Figure 10: Variable quantity of tasks on the system's capacity. Tasks far exceed microserver capacity.

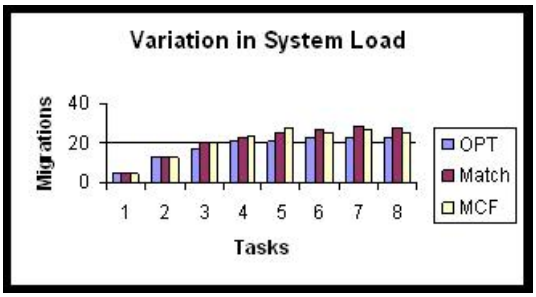

Figure 11: System capacity remains constant while the number of tasks varies. Tasks increase to twice the microserver capacity.

Simulations varying the skew parameter are graphed in Figure 12 and Figure 13. In these simulations, the total time is 12 and the time window W was set equal to the remaining time left until the end of the total time so that the algorithms could make full use of all knowledge about the future. As expected, there is a dramatic improvement in performance as the skew increases since the probabilistic information more accurately reflects the true trajectories of the tasks. The improvement in performance for the multicommodity flow algorithm is more pronounced, decreasing a full 10% from .11 to .47. In the extreme, when full information about the trajectories is known in advance, the multicommodity flow linear program performs exactly the same optimization as the linear program from the deterministic scenario. The simulations suggest that as the certainty in the probabilities increases, the correspondence between the two LP formulations becomes more advantageous.

Variation in Microserver Radius

In Figure 14, there is a spike in the average percentage increase in migrations when the radius is 2. This may be because the problem is most constrained when the radius is around 2, since this is the smallest radius for which the area is mostly covered. The multicommodity flow algorithm performs better in the more constrained, challenging problem settings.

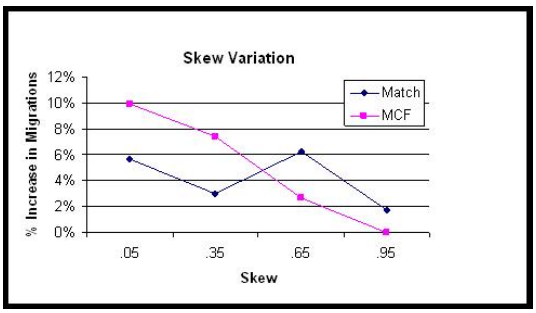

Figure 12: Percentage increase in migrations as a function of variation in skew

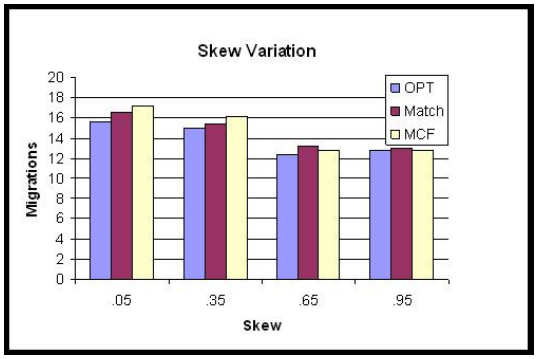

Figure 13: Total migrations as a function of variation in skew

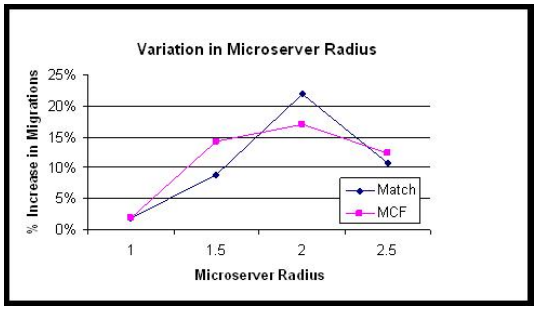

Figure 14: Variation in microserver radius

6. CONCLUSION

We studied the Kinetically Stable Task Assignment problem using two heuristics that take the knowledge of future trajectories into account. The results show that as the knowledge of the trajectories become stronger, both heuristics perform better, but the MCF algorithm improves more than the matching algorithm. When future knowledge is weak, the matching algorithm is a better choice. Since it is less complex and is amenable to distributed implementation, we suggest using the matching algorithm unless the future trajectories of the objects are clear. Also, the matching algorithm is guaranteed to produce an integral solution, whereas the MCF solution may be fractional.

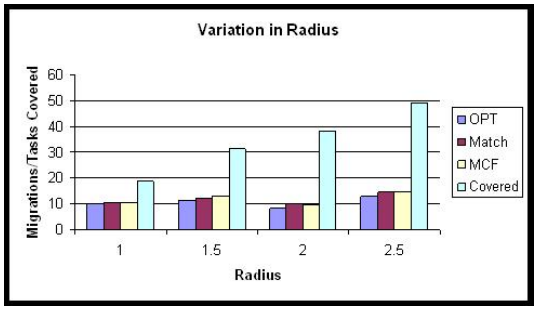

Figure 15: Variation in microserver radius

A particularly attractive option for distributing the matching algorithm over the microservers is the reverse auction for asymmetric assignment problems due to [6, 7]. This auction is well suited for a fast and distributed implementation in our setting because of its accelerated convergence time in simulations [7]. In our context, we do not assume that the tasks have computational power. Therefore, a microserver will serve as a proxy for the tasks it can monitor. For each task there will be some sort of election from amongst the servers that are candidates (perhaps using ID numbers), so that each task is processed by only one server. In the auction algorithm, each task (server) must communicate with every adjacent server (task). With our proxy system, this communication is achieved by a server broadcasting to twice the distance it monitors, so that it reaches all other servers that can monitor the task. Once the proxies have been established, the servers proceed in implementing the forward auction described in [6], assigning tasks to servers. We assume that every server has information about the probability that in the future it will be able to cover the tasks it can currently be assigned to (and therefore the weights of relevant edges is known). When all of server A's tasks have been assigned and no changes made over several time periods, a 'finished' message is sent to the coordinator with the smallest price for every server that has been assigned to a task proxied by A. If during the course of the auction one of the tasks is reassigned to a new server, server A sends a 'processing' message to the coordinator. When the coordinator has received finished messages from all servers and no changes have been made for several time periods, the auction is deemed to have completed. The coordinator now computes the min price over all assigned servers, and broadcasts this value as it is essential knowledge for the next reverse phase of the auction. Once the servers have received the min price, they proceed with the reverse auction. To detect termination, every server sends another 'finished' message to the coordinator once their price is sufficiently low. Note also that the matching problems we are solving at successive time steps are highly correlated (the respective windows overlap in all but one position). We may be able to take advantage of this fact to initialize the auction algorithm so that termination occurs quickly.

We leave as future work the exploration of the auction algorithm or other distributed implementations of the matching algorithm and the multicommodity flow algorithm [4, 2]. There is also future work to be done on the theoretical aspects of the Deterministic Kinetically Stable Matching Algorithm, including refining the integrality gap, explaining why fractional optimal solutions appear only very rarely, and designing algorithms with theoretical approximation guarantees. The probabilistic formulation can also benefit from a deeper understanding of the correlations in the presence of microserver-task edges across successive time steps. We expect that this will further demonstrate the advantage of being able to look ahead and perform microserver assignments taking future positions of the tasks into account.

Acknowledgements The authors wish to thank An Nguyen for many useful discussions. We also thank Nebojsa Jojic for explaining the objective function for the Probabilistic Kinetically Stable Task Assignment problem in terms of a generative model as described in section IV,B. Leonidas Guibas wishes to acknowledge the support of the Microsoft Corporation. Research conducted while Zoë Abrams was visiting Microsoft Research.

7. REFERENCES

[1] N. Kong, and A. Schaefer. A Factor 1/2 Approximation Algorithm for Two-Stage Stochastic Matching Problems.

[2] L. Chattopadhyay, K. Higham, K. Seyffarth. Dynamic and Self-Stabilizing Distributed Matching. PODC2002.

[3] Y. Kopidakis, M. Lamari, V. Zissimopoulos. On the Task Assignment Problem: Two New Efficient Heuristic Algorithms. Academic Press 1997.

[4] D.P. Bertsekas, J.N. Tsitsiklis. Parallel and Distributed Computation: Numerical Methods, Prentice Hall, 1989; republished by Athena Scientific, 1997.

[5] D.P. Bertsekas and J.N. Tsitsiklis, "Some Aspects of Parallel and Distributed Iterative Algorithms - A Survey", Automatica, Vol. 27, No. 1, 1991, pp. 3-21.

[6] D.P. Bertsekas. Auction algorithms for network flow problems: A tutorial introduction. Computational Optimization and Applications, 1:766, 1992.

[7] D.P. Bertsekas, D.A. Castanon, and H. Tsaknakis. "Reverse Auction and the Solution of Inequality Constrained Assignment Problems".

[8] T. Biedl,C. Duncan, R. Fleischer, S. Kobourov. Tight Bounds on Maximal and Maximum Matchings. Discrete Mathematics, volume 285, number 1-3, August 2004 pp7-15.