Collectives for Multiple Resource Job Scheduling Across Heterogeneous Servers

K. Tumer (kagan@email.arc.nasa.gov) and J. Lawson (lawosn@email.arc.nasa.gov)
NASA Ames Research Center
Moffett Field, CA 94035

Abstract. Efficient management of large-scale, distributed data storage and processing systems is a major challenge for many computational applications. Many of these systems are characterized by multi resource tasks processed across a heterogeneous network. Conventional approaches, such as load balancing, work well for centralized, single resource problems, but breakdown in the more general case. In addition, most approaches are often based on heuristics which do not directly attempt to optimize the world utility. In this paper, we propose an agent based control system using the theory of collectives. We configure the servers of our network with agents who make local job scheduling decisions. These decisions are based on local goals which are constructed to be aligned with the objective of optimizing the overall efficiency of the system. We demonstrate that multi-agent systems in which all the agents attempt to optimize the same global utility function (team game) only marginally outperform conventional load balancing. On the other hand, agents configured using collectives outperform both team games and load balancing (by up to four times for the latter), despite their distributed nature and their limited access to information.

Keywords: Reinforcement learning, Job Scheduling, Computational Grid, Multi-resource optimization, Collectives

1. Introduction

In recent years, both the fields of multiagent systems and reinforcement learning have shown great promise in application to large optimization problems [1, 3, 2, 18, 20]. In particular, the intersection of these two fields has produced exciting solutions to many problems (e.g., data routing across a network [10, 12, 14, 16, 19].

Job scheduling, a particular instance of a resource allocation problem, is particularly well suited for a multi agent solution based on reinforcement learning agents [3, 9]. In this problem, we are confronted with a grid of interconnected servers, along with incoming stream of "jobs" or elements that need to be processed. Potentially, each server in the grid has different capabilities, and not all jobs can (or should be) processed at the servers in which they enter the system.

For the single-resource case, this problem has been extensively studied [9]. However, multi-resource job scheduling across a network of heterogeneous servers is a difficult problem that has received much less attention [7]. In this instance of the problem, each server has multiple, and potentially heterogeneous, resource capabilities (e.g., CPU speed, available memory, queue length); and each incoming job to be processed has different characteristics across those resources as well. [1]

Load balancing (LB) has been successfully applied to single resource scheduling problems. In fact, for single resource optimization problems, there are theoretical results showing that load balancing does provides optimal solutions [9].

Generalizing LB to the multi-resource case, though, is far from straightforward. In its simplest form, load balancing aims at ensuring that the level of activity on each server stays the same, i.e., the load on the system is balanced across all the servers. Load balancing though has two major limitations. First, it requires centralized control.[2] Second, load balancing assumes that the load being distributed across the servers is a de-facto desirable solution to optimizing the world utility problem. In the multi-resource case, these issues become even more problematic as getting maximal efficiency not only of the system, but also of the utilization of the available resources must be addressed. Different generalizations have tended to emphasize different desirable characteristics [7].

The agent based approach we propose sidesteps this potential mismatch between balancing the load across the network and optimizing the world utility function. It directly aims to optimize the world utility and as a consequence it is possible that some servers are idle while others are operating at full capacity. As long as that system behavior is good for the world utility, no consideration is being made to "split" the load or make the jobs processing "fair" in any way.

There are many possible ways to map this problem onto a multi-agent system. One possibility would be to let the jobs be agents with their actions being the choice of server on which to run. Another possibility, and the one we will consider, is to assign many agents to each server. These agents are tasked with the problem of deciding which jobs in the local wait queue will be run locally and which should be shipped to another server for processing. One choice for the action space of these agents is to select the particular neighbor (including

[1] Throughout this paper, we refer to servers with different resource configurations as "heterogeneous" servers. We assume that there are no compatibility issues related to compilation of the jobs, and that any job can be executed at any server, assuming the server has the necessary resources. In some articles [7], this type of network is referred to as a "near-homogeneous" computational grid.

[2] There are LB algorithms based on local information, though the performance of such algorithms necessarily suffers from the loss of information [7].

itself potentially) to whom to ship a given job. In some situations, it might be expected that an agent might begin preferentially shipping all or most of its jobs to the same neighbor. This could lead to congestion difficulties for the system. To avoid this possibility, we instead assign to each agent a vector $\vec{p}$ whose components give the probability of routing a job to its various neighbors. In this scenario, the agents are given the more abstract job of setting their own probability vector. The design question now becomes to determine what rewards each agent should receive so that they set the probability vectors that optimize the overall job processing efficiency of the full system.

Traditional solutions to that question include the "team game" approach, where each agent receives the full world reward, and the "selfish reward" where each agent is only concerned about the jobs that it has touched. In general, team game solutions suffer from the signal-to-noise problem in which an agent has a difficult time discerning the effects on its actions on its utility, because that "signal" is getting swamped by the "noise" of all the other agents. Clearly this problem gets worse as the number of agents in a system increases. Selfish utilities on the other hand suffer from coordination issues, where actions that may be beneficial to one agent may cause significant damage to the system overall. In other words, there are no guarantees that agents using selfish utilities will act in the best interests of the overall system.

Handtailored solutions may in some cases outperform these generic utilities, but such solutions though appealing generally:

- require laborious modeling;
- provide "brittle" global performance;
- are not "adaptive" to changes in the environment; and
- generally do not scale well.

The theory of collectives is concerned with overcoming the shortcomings of team games and selfish utilities without resorting to handtailoring. [3] In particular, it is concerned with providing agents with with rewards that are both "learnable" i.e., they have good signal-to-noise ratios, and are "factored" i.e., the utilities are aligned with the world utility.

A naturally occurring example of a system that can be viewed as a collective is a human economy. Each individual trying to maximize

[3] A collective is defined as a multi agent system in which there is a well-defined world utility function that needs to be optimized, and where each agent takes actions based on its own private utility [15].

their own private utilities (e.g., maximize bank account, advance career) constitute the "agents" in the system. The world utility can be viewed as the gross domestic product of the country in question ("world utility" is not a construction internal to a human economy, but rather something defined from the outside). The issue in such a case is to determine what each agent needs to do so that the joint action of all agents optimizes sthe world utility.

This system needs to be factored to avoid phenomena such as the tragedy of the commons, in which individual avarice works to lower world utility [6], from occurring. One way to avoid such phenomena is by modifying the agents' utility functions via punitive legislation, in essence making sure the agents' utility functions are aligned with the world utility. Securities and Exchange Commission (SEC) regulations designed to prevent insider trading can be viewed as a real world example of an attempt to make such a modification to the agents' utilities.

In designing a collective we have more freedom than the SEC though, in that there is no base-line "organic" private utility function over which we must superimpose legislation-like incentives. The entire "psychology" of the individual agents is at our disposal, and that freedom is a major strength of the collectives approach. For example, it obviates the need for honesty-elicitation mechanisms, like auctions, which form a central component of conventional economics.

The problem of designing collectives is related to work in many fields beyond multiagent systems and computational economics, including mechanism design, reinforcement learning for adaptive control, computational ecologies, and game theory. However none of these fields directly addresses the inverse problem of how to design the agents' utilities to reach a desirable world utility value in its full generality. This is even true for the field of mechanism design, which while addressing an inverse problem similar to that of COIN design, does so only for certain restricted domains, and does not address the "learnability" issue. (Mechanism design is mostly appropriate when there are prespecified goals underlying agents' utilities over which "incentives" need to be provided, and when Pareto-optimality (rather than optimization of a world utility) is often the goal [15].)

The collectives framework has been successfully applied to multiple domains including packet routing over a data network [12], the congestion game known as Arthur's El Farol Bar problem [17], and multi-rover coordination where agents needed to learn sequences of actions to optimize the world utility [11]. Furthermore, in the routing domain, the COIN approach achieved performance improvements of a factor of three over the conventional Shortest Path Algorithm (SPA) routing algorithms currently running on the internet [14], and avoided the Braess' routing paradox which plagues the SPA-based systems [12].

In this paper we present a general, distributed reinforcement learning solution to such optimization problems. In Section 2 we summarize the theory of collectives that is relevant to this application. In Section 3 we present the details of this domain, and show how the collectives approach can be used for job scheduling. In Section 4, we show simulation results demonstrating the superiority of the collective-based approach, where the multi agent system approach significantly outperforms load balancing, even though it has less information at its disposal.

2. Collectives: A Summary

In this section, we summarize the portion of the theory of collectives required for the experiments described in this article [15]. Let Z be an arbitrary vector space whose elements z give the joint move of all agents in the system (i.e., z specifies the full "worldline" consisting of the actions/states of all the agents). The provided **world utility** $G(z)$, is a function of the full worldline, and the problem at hand is to find the z that maximizes $G(z)$.

In addition to G, for each agent η, there is a **private utility functions** $\{g_\eta\}$. The agents act to improve their individual private functions, even though, we, as system designers are only concerned with the value of the world utility G. To specify all agents other than η, we will use the notation $\hat{}\eta$.

Our uncertainty concerning the behavior of the system is reflected in a probability distribution over Z. Our ability to control the system consists of setting the value of some characteristic of the collection of agents, e.g., setting the private functions of the agents. Indicating that value by s, our analysis revolves around the following central equation for $P(G \mid s)$, which follows from Bayes' theorem:

$$P(G \mid s) = \int d\vec{\epsilon}_G P(G \mid \vec{\epsilon}_G, s) \int d\vec{\epsilon}_g P(\vec{\epsilon}_G \mid \vec{\epsilon}_g, s) P(\vec{\epsilon}_g \mid s) \; , \qquad (1)$$

where $\vec{\epsilon}_g$ is the vector of the "intelligences" of the agents with respect to their associated private functions, and $\vec{\epsilon}_G$ is the vector of the intelligences of the agents with respect to G. Intuitively, what these vectors indicate what percentage of η's actions would have resulted in lower utility. In this article, we use intelligence vectors as decomposition variables for Equation 1 (see [15] for details on intelligence).

Note that, from a game-theoretic perspective, a point z where all players are rational, ($\epsilon_{g_\eta} = 1$ for all agents η, is a game theory Nash equilibrium [15]. On the other hand, a z at which all components of $\vec{\epsilon}_G = 1$ is a local maximum of G (or more precisely, a critical point of the $G(z)$ surface).

The design of collective problem can be best illustrated by the trade-off presented in Equation 1. If we can choose s so that the third conditional probability in the integrand, $P(\vec{\epsilon}_g \mid s)$, is peaked around vectors $\vec{\epsilon}_g$ all of whose components are close to 1 (that is agents are able to "learn" their tasks), then we have likely induced large private utility intelligences. If we can also have the second term, $P(\vec{\epsilon}_G \mid \vec{\epsilon}_g, s)$, be peaked about $\vec{\epsilon}_G$ equal to $\vec{\epsilon}_g$ (that is the private and world utilities are aligned), then $\vec{\epsilon}_G$ will also be large. Finally, if the first term in the integrand, $P(G \mid \vec{\epsilon}_G, s)$, is peaked about high G when $\vec{\epsilon}_G$ is large, then our choice of s will likely result in high G, as desired. Note, this first term requires global information (search for global optima, rather than local optima). In problems where such communication/control is not possible, only terms two and three can be optimized.

2.1. Factoredness and Learnability

The requirement that private functions have high "signal-to-noise" (an issue not considered in conventional work in mechanism design) arises in the third term. It is in the second term that the requirement that the private functions be "aligned with G" arises. In this work we concentrate on these two terms, and show how to simultaneously set them to have the desired form.

Details of the stochastic environment in which the collection of agents operate, together with details of the learning algorithms of the agents, are reflected in the distribution $P(z)$ which underlies the distributions appearing in Equation 1. Note though that *independent of these considerations*, our desired form for the second term in Equation 1 is assured if we have chosen private utilities such that $\vec{\epsilon}_g$ equals $\vec{\epsilon}_G$ exactly *for all z*. We call such a system **factored**. In game theory language, the Nash equilibria of a factored system are local maxima of G. In addition to this desirable equilibrium behavior, factored systems also automatically provide appropriate off-equilibrium incentives to the agents (an issue rarely considered in the game theory / mechanism design literature).

As a trivial example, any "team game" in which all the private functions equal G is factored [4]. However team games often have very poor forms for term 3 in Equation 1, forms which get progressively worse as the size of the system grows. This is because for large systems where G sensitively depends on all components of the system, each agent may experience difficulty discerning the effects of its actions on

$$\begin{array}{c} \\ \eta_1 \\ \eta_2 \\ \eta_3 \\ \eta_4 \end{array} \begin{array}{c} z \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \end{array} \quad \begin{array}{c} \Longrightarrow \\ \text{Clamp } \eta_2 \\ \text{to ``null''} \end{array} \quad \begin{array}{c} (z_{\hat{\eta}_2}, \vec{0}) \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \end{array}$$

Figure 1. This example shows the impact of the clamping operation on the joint state of a four-agent system where each agent has three possible actions, and each such action is represented by a three-dimensional unary vector. The first matrix represents the joint state of the system z where agent 1 has selected action 1, agent 2 has selected action 3, agent 3 has selected action 1 and agent 4 has selected action 2. The second matrix displays the effect of clamping agent 2's action to the "null" vector (i.e., replacing z_{η_2} with $\vec{0}$).

G. As a consequence, each η may have difficulty achieving high g_η in a team game. We can quantify this signal/noise effect by comparing the ramifications on $g_\eta(z)$ arising from changes to z_η with the ramifications arising from changes to $z_{\hat{\eta}}$ (i.e., changes to all nodes *other* than η). In particular, the **learnability** of private utility g_η gives the ratio of the sensitivity of $g_\eta(z)$ is to changes to agents other than η, to the sensitivity of $g_\eta(z)$ to changes to η. So at a given state z [15]. the higher the learnability, the more $g_\eta(z)$ depends on the move of agent η, i.e., the better the associated signal-to-noise ratio for η. Intuitively then, higher learnability means it is easier for η to achieve a large value of its intelligence.

2.2. Private Utilities

As discussed above, designing the private utilities for the agents is one of the main challenges in a collective. One private utility function which is factored and generally provides good learnability is the **Wonderful Life** Utility (WLU) [15, 17, 12]. The WLU for agent η is parameterized by a pre-fixed **clamping parameter** CL_η chosen from among η's legal or illegal moves:

$$WLU_\eta \equiv G(z) - G(z_{\hat{\eta}}, CL_\eta) . \qquad (2)$$

Figure 1 provides an example of clamping. As in that example, in many circumstances there is a particular choice of clamping parameter for agent η that is a "null" move for that agent, equivalent to removing that agent from the system, hence the name of this private function. For such a clamping parameter WLU is closely related to the economics technique of "endogenizing a player's (agent's) externalities" [8]. Indeed, WLU has conceptual similarities to Vickrey tolls [13] in economics, and Groves' mechanism [5] in mechanism design. However, because WLU can be applied to arbitrary, time-extended utility functions, and need not be restricted to the "null" clamping operator interpretable in terms of "externality payments", it can be viewed a generalization of these concepts.

It can be proven that in many circumstances, especially in large problems, that WLU has higher learnability than does the team game choice of private utilities [15]. This is mainly due to the second term of WLU which removes a lot of the effect of other agents (i.e., noise) from η's utility. The result is that convergence to optimal G with WLU is much quicker (up to orders of magnitude so [15]) than with a team game.

Intuitively, one can look at WLU from the perspective of a human company, with G the "bottom line" of the company, the agents η identified with the employees of that company, and the associated g_η given by the employees' performance-based compensation packages. For example, for a "factored company", each employee's compensation package contains incentives designed such that the better the bottom line of the corporation, the greater the employee's compensation. As an example, the CEO of a company wishing to have the private utilities of the employees be factored with G may give stock options to the employees. The net effect of this action is to ensure that what is good for the employee is also good for the company. In addition, if the compensation packages have "high learnability", the employees will have a relatively easy time discerning the relationship between their behavior and their compensation. In such a case the employees will both have the incentive to help the company and be able to determine how best to do so. Note that in practice, providing stock options is usually more effective in small companies than in large ones. This makes perfect sense in terms of the COIN formalism, since such options generally have higher learnability in small companies than they do in large companies, in which each employee has a hard time seeing how his/her moves affect the company's stock price.

3. Collectives for Multi-Resource Optimization

With increasing demand for supercomputing resources (e.g., biological applications), the ability of a system to efficiently schedule and process jobs is becoming increasingly important. As such, heterogeneous computational grids where jobs can enter the network from any point and be processed at any point are becoming increasingly popular. Below, we discuss such a grid of computational servers, and show how the theory of collectives summarized above can be applied to this problem.

3.1. System Model

We modeled such a computational system as a network of N servers each with K resources $(r_1, ...r_k)$. Each server had a specified capacity for each resource assigned to be an integer ranging from $[1, M]$. Thus, M was a measure of the heterogeneity of the resources. The first resource r_1 corresponded to the processing speed of the server. The other resources corresponded to other, and perhaps limited, quantities, such as memory. In general, each server had 2-4 neighbors with which it had a direct connection.

Jobs were also specified by K resource requirements ranging from $[1, M]$. The first job resource r_1 was an indication of the number of cycles the job required to be processed. The other resources, again, corresponded to other quantities, such as memory. An important point is that for resources $r_i, i > 1$, the server resource capacity must equal or be greater than the job's requirement in order for a job to run on a particular server. This could correspond to the requirement that a server must have enough memory to accommodate a given job.

Each server had its own wait queue for jobs. For simplicity, we allowed only one job to run on a server at a time; the other jobs remained in the queue until the processor became available. Jobs entered the local queues either externally (to the system) or were shipped from other servers. Jobs entering externally were sent to the back of the queue while jobs received from other queues go to the front. Shipped jobs go to the front for two reasons. First, they have already had to wait in the queue they were originally placed. Secondly, shipped jobs are often "difficult" jobs in the sense of finding an appropriate server to run them. Putting them in the front of the queue forces the system to deal with these jobs now rather than postponing action by having them wait in another queue. Otherwise, it would be easy to imagine difficult jobs being endlessly shuffled.

If the processor was available, and the resource requirements met, the server would activated the first job in the queue. If the processor was available, but the server did not have the resource capacity to run the job, the server would remain idle until the problem job was sent to another server. This is expected to be one the main causes of bottlenecks in the system and will be an issue that an intelligent job management system will need to address.

The dynamics of our simulations thus proceeded as follows. At each time step τ, new jobs were added to the system and placed in the wait queue of randomly selected servers. In particular, each server had a probability r of receiving a new job at each time. If a given processor was idle, and the first job in the queue met the resource requirements, that job would be activated. If not, the server would remain idle. In addition, for each τ, the server would make a decision about the first job in the queue, deciding whether to keep the job or sent it to a neighboring server. These decisions were made based on the agents' probability vectors which in turn are set using reinforcement learning algorithms. This will be discussed in more detail below.

Thus, there were two main sources of inefficiency in the system. The first were the bottlenecks created by jobs whose requirements exceeded the capacity of their server. When such a job got to the front of the queue, the server remained idle until the job was shipped to a neighbor. The second source of inefficiency arose from mismatches between a processor's speed and a job's cycle requirement. Clearly, jobs that require more cycles should run on faster machines.

3.2. Multi-Agent Architecture

These inefficiencies were the main issues that agents as shipping decision makers needed to manage. The heterogeneity of both the servers and the jobs resulted in many possible combinations of assignments of jobs to servers. This was especially true as M, the resource range, grew, and could potentially create a very noise environment in which the agents had to learn. To reduce this noise, we instead assigned multiple agents to each server where each agent dealt with a subset of jobs. In particular, for jobs with K resources we assigned 2^K agents per server where agent 1 deals with jobs such that $r_1 \epsilon [1, M/2], ..., r_k \epsilon [1, M/2]$, agent 2 deals with jobs $r_1 \epsilon [M/2 + 1, M], r_2 \epsilon [1, M/2] ..., r_k \epsilon [1, M/2]$, etc. Thus the resource specifications of a job determined which agent would make its shipping decision.

We will distinguish between two time scales that will be used throughout this article: τ gives the time steps at which the jobs enter the system, move between queues, and are processed, whereas t gives the time steps at which the agents observe their utilities, change their actions, etc. This distinction is important because it is the only way an agent can get a "signal" from the system that will reflect the impact of its decision, i.e, the system has to settle down before a reward can be matched to an action. Therefore, an agent η changes its probability vector at each time t. Withing a single time step t though, many jobs enter the system, are executed, routed etc. each of which occurs at time interval τ $(t >> \tau)$.

The learning was organized as follows. For each t, the probability vectors were fixed, and the simulation run for fixed number of time steps (typically, 400). At the end of this run, the utility functions were calculated and the rewards recorded in the agents' training sets. In order to be able to compare the performance individual probability vectors, we cleared the system (i.e. the queues) after each t. During the initial phase, $0 \leq t \leq 100$, the proability vectors were set at random, and the utilities recorded. After this "data collection" phase, $t \geq 100$, the agents utilized reinforcement learning algorithms to set their probability vectors.

The learning algorithm proceeds by first generating a set of candidate probability vectors with a Gaussian distribution about the current probability vector. Reward estimates were made by performing a weighted average over reward values from the agents' training set. These values were weighted by both how long ago the value was recorded (data aging) and the distance between the candidate and the previous probability vector

$$R = \Sigma_i g_i e^{-dT_i} e^{-dP_i} / \Sigma_i e^{-dT_i} e^{-dP_i}. \tag{3}$$

Here, $dT_i = \alpha_T(T - t_i)$ where T is the current learning period, t_i is the period for data g_i, and α_T is a parameter. Also, $dP_i = \alpha_P||\vec{P} - \vec{p_i}||$ where $\vec{P}$ is the current probability vector, $\vec{p_i}$ is the vector for data g_i, and α_P is a parameter. The new probability vector was then chosen by sampling a Boltzmann probability distribution over these reward estimates.

3.3. State Space and World Utility

Let us define the state of each agent at time t as by

$$z_{\eta,t} = \{(j_0, w_0, I_0^{\eta,t}, e_0^{\eta,t}), \cdots, (j_k, w_k, I_k^{\eta,t}, e_k^{\eta,t}), \cdots\} \tag{4}$$

where j_k identifies job k, w_k is the weight of that job which gives the importance of that job in the system, $I_k^{\eta,t}$ is the "job indicator" function and is equal to 1 if job k was "touched" by agent η at time step t, and 0 otherwise, and $e_k^{\eta,t}$ determines whether job k was executed at agent η at time step t.

Now, the state of the full system, z_t at time t, is given by:

$$z_t = \{(j_0, w_0, 1, e_0^t), \cdots, (j_k, w_k, 1, e_k^t), \cdots\} \tag{5}$$

where e_k^t determines whether job k was executed at time step t. Note that the job indicator function I_k^t is always set at 1 for the full system, since by definition, if the job is in the system, it must have been "touched" by at least one agent. Nevertheless, we keep the notation, both for ensuring consistency between the state vector of an agent and that of the full system, and because its presence in the world utility will facilitate the derivation of the private utilities of the agents.

Based on this, the world utility at time t is given by:

$$G(z_t) = \frac{\sum_k w_k.e_k^t}{\sum_k w_k^t} \tag{6}$$

Intuitively, G gives the weighted ratio of the all the jobs that were processed at time step t to all jobs that entered the system at that time step (recall that "time step t is a window of time, not a single time step from the point of view of the jobs".)

For all the reasons highlighted in Section 2, using G as the reward for all the agents introduces significant signal to noise issues. In order to overcome such difficulties, we explored the use the of collective based private utilities discussed in 2. In particular, we investigate the case where the clamping parameter set to the null vector. This corresponds to $I_k^{\eta,t}$ being set to 0 for all jobs k for which it was set to 1 at time step t. With this choice for clamping, the WLU is given by:

$$\begin{aligned} WLU(z_{\eta,t}) &= G(z) - G(z_{\hat{\eta}}, (CL_\eta)) \\ &= \frac{\sum_k w_k.e_k^t}{\sum_k w_k^t} - \frac{\sum_k w_k.e_k^t.\bar{I}_k^{\eta,t}}{\sum_k w_k^t} \\ &= \frac{\sum_k w_k.e_k^t.I_k^{\eta,t}}{\sum_k w_k^t} \end{aligned} \tag{7}$$

$$\tag{8}$$

where $\bar{I}_k^{\eta,t}$ is the complement of $I_k^{\eta,t}$ and equals 1 when $I_k^{\eta,t}$ equals 0 and 0 when $I_k^{\eta,t}$ equals 1. Intuitively, $WLU(z_\eta)$ represents the weighted fraction of jobs that were touched by agent η to the jobs that entered the system. Note, this is different than what a "selfish utility" (SU) only concerned with its own jobs would do. More precisely, let us define such a utility:

$$SU(z_{\eta,t}) = \frac{\sum_k w_k.e_k^t.I_k^{\eta,t}}{\sum_k w_k^t.I_k^{\eta,t}} \tag{9}$$

Intuitively, SU gives the ratio of the jobs processed by the system at time step t, to the total jobs that passed through that agent, hence the indicator function in the denominator. In the language of collectives, this utility has higher learnability than does the WLU, but it is not factored with G. The impact of this tradeoff is explored below in Section 4.

3.4. Load Balancing Algorithm

We compared our agent-based approach against a fixed, deterministic algorithm. In particular, we considered a distributed version of multi-resource load balancing. For each server, we calculated a load for each of the k resources, $l_k = \Sigma_n^N(j_k^n/s_k)$ where j_k^n is the resource k of job n and s_k is the the capacity of resource k of the server. Thus, the resource load has been normalized to the resource capacity of the server. We assign a load to a particular server i as the average of its individual resource loads $L_i = Avg(l_k)$. We, then, calculate the system load as the average over the servers $L_{avg} = Avg(L_i)$.

The load balancing algorithm proceeds as follows. At each time step τ, each server calculates its own load and compares it with the global load L_{avg}. If the server's load is greater than the global, modulo some tolerance, the servers looks to get rid of its highest load job. Each server has access to global information about the loads on the all the other servers. Using this information, the server determines which of the other servers has the lowest load. It then ships its high load job to the low load server via the one of its neighbors that lies on the shortest path between the sending and the receiving servers.

Table I. System Processing Efficiency (r=0.2,M=2)

Algorithm	Net Efficiency	Perc Gain
Opt Estimate	1.0	-
RAND	0.9318	-
SU	0.9317	-0.20%
TG	0.9470	22.33%
WLU	0.9788	68.83%
LB	0.9971	95.70%

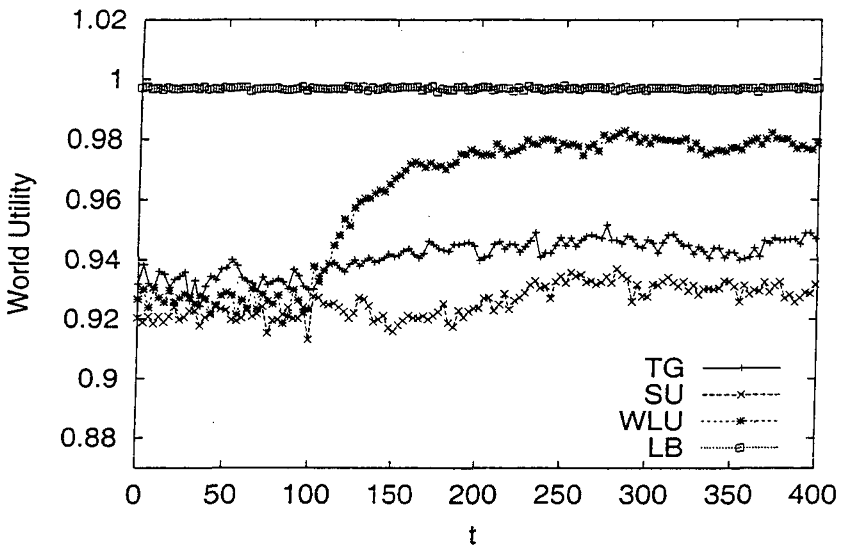

Figure 2. Simulations results for 50 servers with 4 agents each with parameter values (r=0.2,M=2). Each t represents a "run" of 400 τ time steps with each agent having a fixed probability vector $\vec{p}$ during the run. At the end of each run, utilities are calculated, the queues cleared, and the agents reset/modify their $\vec{p}$ based on their learning algorithms. Results are averages over 50 different systems configurations.

Table II. System Processing Efficiency (r=0.2,M=8)

Algorithm	Net Efficiency	Perc Gain
Opt Estimate	1.0	-
RAND	0.6435	-
SU	0.6345	-2.53%
TG	0.6703	7.51%
WLU	0.7932	41.97%
LB	0.2254	-117.28%

4. Results

We ran extensive simulations on networks of $N = 50$ servers having $K = 2$ resources. The 50 servers had 4 agents each, making for 200 total agents in the system. The servers were connected into a network having a ring configuration with random connections added in the spirit

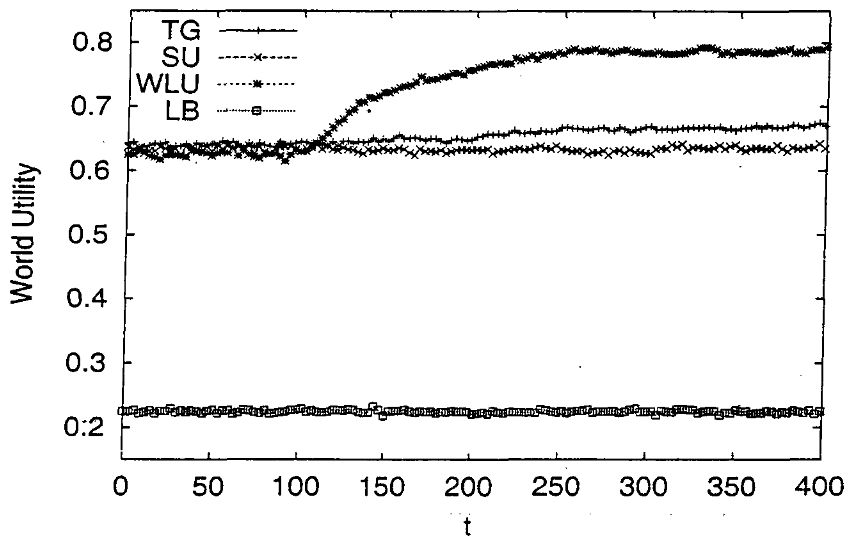

Figure 3. Simulations results for 50 servers with 4 agents each with parameter values (r=0.2,M=8). Load balancing does especially poorly in cases of large heterogeneity $M = 8$ due to its inability to deal effectively with bottlenecks. In learning based methods TG, SU, WLU, agents set probability vectors randomly for $t \leq 100$ as part of their training. Even the training period RAND performance is better than load balancing.

of "small world's" networks. In general, each server had 2-4 neighbors with which it had a direct connection.

We examined the performance for different job arrival probabilities r and different resource ranges M. We tabulated the performance for

Table III. System Processing Efficiency (r=0.8,M=2)

Algorithm	Net Efficiency	Perc Gain
Opt Estimate	0.781	-
RAND	0.6260	-
SU	0.6140	-7.78%
TG	0.6376	7.48%
WLU	0.6911	41.98%
LB	0.6446	11.97%

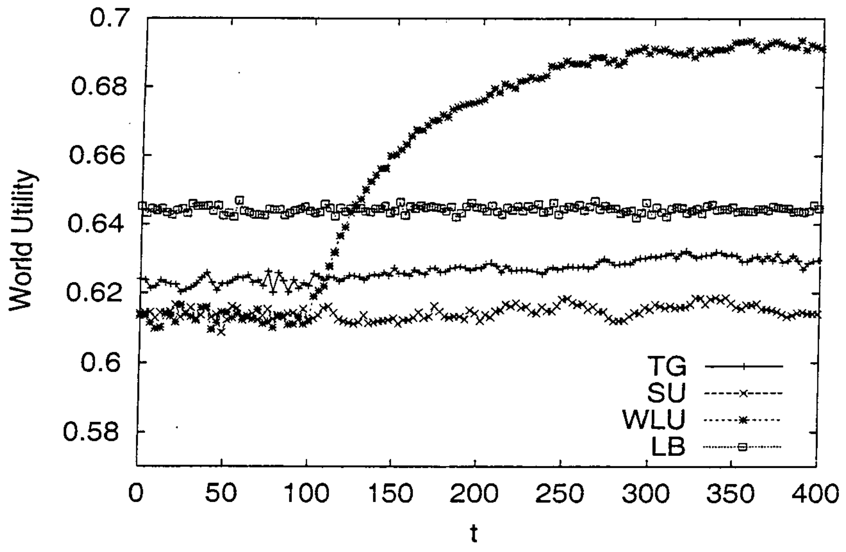

Figure 4. Simulations results for 50 servers with 4 agents each with parameter values (r=0.8,M=2).

Table IV. System Processing Efficiency (r=0.8,M=8)

Algorithm	Net Efficiency	Perc Gain
Opt Estimate	0.395	-
RAND	0.1944	-
SU	0.2024	4.01%
TG	0.1984	1.98%
WLU	0.2490	27.25%
LB	0.0974	-48.32%

the multi-agent approach with learning agents, a load balancing algorithm generalized for the multi-resource case, and a random shipping algorithm RAND. In the RAND algorithm, the proportion vectors for shipping/holding the first job in the queue was set randomly. This is is basically the situation when the agents are in the training phase of their learning algorithm. For scenarios involving learning agents, we did runs using personal utilities based on team games (TG), selfish agents

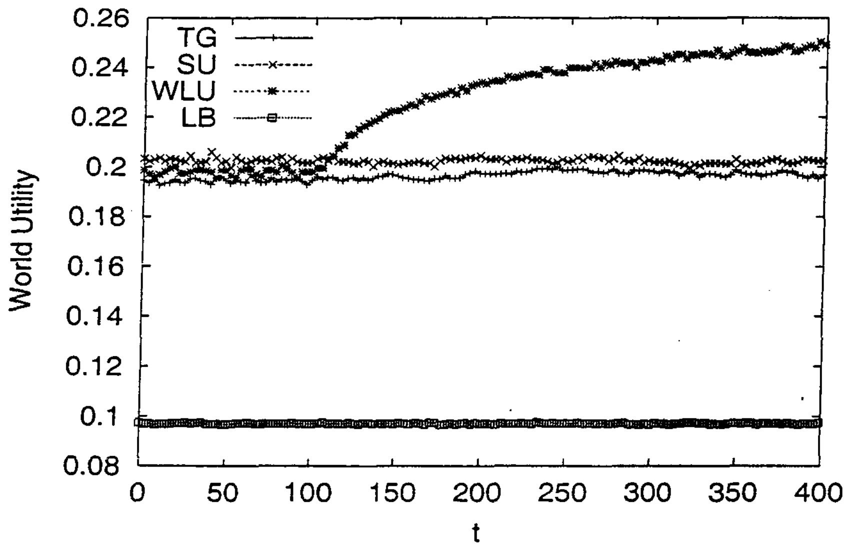

Figure 5. Simulations results for 50 servers with 4 agents each with parameter values (r=0.8,M=8).

(SU), and theory of collectives (WLU). The results were averaged over 50 different randomly generated network configurations.

Certain parameter values, especially for large heterogeneity (large M), introduced a large amount of frustration into the system. In these cases, it would be impossible for the system to achieve 100% processing efficiency. For these cases, we made an estimate of what we might expect the theoretical optimal possible performance to be. These estimates are included in the tables. To further compare the results, we calculated the performance *gain* of the different methods. This gain is measured relative to the gap between random shipping and a theoretical upper bound on the performance of the system.

We obtained this bound by first analyzing what percentage of incoming jobs can be processed at their point of entry into the system, if the incoming rate were set to one (i.e., $r = 1$), meaning each server receives a job at each time step. In such cases, no job shifting can take place, since each server simply processes the jobs it receives. Then, for other values of r, we assumed instantaneous shipping across the servers, allowing the job that cannot be processed at their point of entry to reach servers in which they can be processed. This is not a particularly tight bound since it ignores how the "slack" in the system picks up the unprocessed jobs (i.e., ignores how a server with high capacity will receive and schedule these jobs), and simply assumes that if there is room at some server, the jobs will appear there and be processed.

Tables I-IV show the absolute and relative performance numbers for the different algorithms at $t = 400$. Notice that in all cases the learning based approaches are competitive or significantly outperform load balancing. Load balancing does well for low arrival rates r and low heterogeneity M. But its performance degrades markedly for high r, and especially for high M. In fact, even setting the probability vectors at random (RAND) outperforms load balancing for $M = 8$ independent of r. This can be understood by the fact that the agent based approaches make decisions about only the first job in the queue. But it is this first job that can create serious bottlenecks in the system; if the first job needs more resources than the server can provide, the job cannot run and remains in queue, blocking other jobs from being processed as well. Load balancing, on the other hand, is attempting only to equalize the load across on the entire queue and does nothing to deal with such potential bottlenecks. For large M, the potential for bottlenecks increases markedly. Random probability vectors have the advantage over load balancing that they operate directly on the place where a bottleneck can occur.

It is also in these large M regimes that approaches based on adaptive learning algorithms would be expected to do well. Simulations results show large increases in performance by having the probability vectors set using reinforcement learning. These results also show the importance of setting the agents' personal utilities to be functions that are both "factored" and "learnable". The team game (TG) utility is factored trivially, but has poor learning properties for the individual agents since it includes information from the full system. The selfish (SU) utility is expected to be more learnable since it only includes effects of individual agents, but it is not factored (aligned with the global goal), and therefore could be doing a good job of learning the wrong thing. The Wonderful Life (WLU) utility derived using the theory of collectives is both factored and learnable. It consistently outperforms TG and SU for all parameter pairs (r, M). Figures 2-5 provide the results for two r and two m combinations. In addition, WLU outperforms load balancing in all but the simplest case. The performance gap is especially large for the $M = 8$ simulations, where WLU outperforms load balancing by a factor of 2-4.

5. Conclusions

In this work we investigated how a collective of reinforcement learning agents can learn to effectively solve a multi-resource optimization problem. In particular we focus on the multi-resource job scheduling problem across a heterogeneous network. Conventional approaches to such problems (e.g., as load balancing) work well when there is instantaneous, centralized control. For all but very few applications, this is an unreasonable assumption on the system's capabilities. Practical, heuristics based approaches on the other hand provide good solutions for the resource problems, but often break down in the more general, multi-resource optimization case.

The collective based solution we propose is based on assigning agents to each server whose actions are to determine whether a job should be processed at that server or shipped to another agent, and if so, to which server. These decisions are based on private utility functions (i.e., local goals) which are constructed to be aligned with the world utility (i.e., optimizing the overall efficiency of the system).

Our results demonstrate that in a collective in which all the agents attempt to optimize the same global utility function (team game) only provide marginal improvements over conventional load balancing. However, those marginal improvements are obtained without requiring a centralized controller (only requirement is of world utility being broadcast at regular intervals). Furthermore, agents using private utility functions based on the theory of collectives outperform both team games and load balancing (up to four times), despite requiring less information.

6. Acknowledgments

The authors would like to thank David Wolpert for helpful discussions.

References

1. J. A. Boyan and M. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. In *Advances in Neural Information Processing Systems - 6*, pages 671–678. Morgan Kaufman, 1994.
2. J. A. Boyan and A. Moore. Learning evaluation functions for global optimization and boolean satisfiability. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence.* AAAI Press, 1998.
3. J. Bredin, R.T. Maheswaran, C. Imer, T. Basar, D. Kotz, and D. Rus. A game-theoretic formulation of multi-agent resource allocation. In *Proceedings of the fourth International Conference of Autonomous Agents*, pages 349–356, 2000.
4. R. H. Crites and A. G. Barto. Improving elevator performance using reinforcement learning. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems - 8*, pages 1017–1023. MIT Press, 1996.
5. T. Groves. Incentives in teams. *Econometrica*, 41:617–631, 1973.

6. G. Hardin. The tragedy of the commons. *Science*, 162:1243–1248, 1968.
7. W. Leinberger, G. Karypis, V. Kumar, and R. Biswas. Load balancing across near-homogeneous multi-resource servers. In *Proceedings of the ninth heterogeneous Computing Workshop*, pages 61–70, Cancun, Mexico, 2000.
8. W. Nicholson. *Microeconomic Theory.* The Dryden Press, seventh edition, 1998.
9. B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems.* IEEE Computer Society Press, 1995.
10. P. Stone. TPOT-RL applied to network routing. In *Proceedings of the Seventeenth International Machine Learning Conference*, pages 935–942. Morgan Kauffman, 2000.
11. K. Tumer, A. Agogino, and D. Wolpert. Learning sequences of actions in collectives of autonomous agents. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 378–385, Bologna, Italy, July 2002.
12. K. Tumer and D. H. Wolpert. Collective intelligence and Braess' paradox. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 104–109, Austin, TX, 2000.
13. W. Vickrey. Counterspeculation, auctions and competitive sealed tenders. *Journal of Finance*, 16:8–37, 1961.
14. D. H. Wolpert, S. Kirshner, C. J. Merz, and K. Tumer. Adaptivity in agent-based routing for data networks. In *Proceedings of the fourth International Conference of Autonomous Agents*, pages 396–403, 2000.
15. D. H. Wolpert and K. Tumer. Optimal payoff functions for members of collectives. *Advances in Complex Systems*, 4(2/3):265–279, 2001.
16. D. H. Wolpert, K. Tumer, and J. Frank. Using collective intelligence to route internet traffic. In *Advances in Neural Information Processing Systems - 11*, pages 952–958. MIT Press, 1999.
17. D. H. Wolpert, K. Wheeler, and K. Tumer. Collective intelligence for control of distributed dynamical systems. *Europhysics Letters*, 49(6), March 2000.
18. M.-J. Yoo. An industrial application of agents for dynamic planning and scheduling. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 264–271, Bologna, Italy, July 2002.
19. W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1114–1120, 1995.
20. W. Zhang and T. G. Dietterich. Solving combinatorial optimization tasks by reinforcement learning: A general methodology applied to resource-constrained scheduling. *Journal of Artificial Intelligence Research*, 2000.