# Resource Allocation Using Fine-Grained Demand Models

**Kutluhan Erol and Robert Kohout**

Intelligent Automation Inc.
2 Research Place #202
Rockville MD, 20850
kutluhan@i-a-i.com, kohout@i-a-i.com

**Abstract**

Multi-agent problem solving using market mechanisms is a primary focus area of Intelligent Automation Inc. We have been developing agent-based planning and scheduling techniques, and investigating their feasibility in manufacturing and transportation domains. These domains involve online streams of jobs being executed in dynamic and uncertain environments. Many NASA applications, such as scheduling DSN antenna facilities, and mission planning share these characteristics. We believe that market mechanisms greatly facilitate problem solving under these conditions, insofar as they provide a means of localizing decision making. This paper describes how we use fine-grained models of cost to facilitate computational resource allocation in Cybele, an agent infrastructure that we have been developing to support multi-agent processing on a distributed network of computers. This technique of resource allocation using fine-grained demand models may be of utility in finding high quality solutions to the DSN scheduling problem.

## The NASA Demand-Based Autonomous Network Scheduling System

(Chien,et.al, 1997b) describes the DANS rescheduling and resource allocation system, which is designed to support incremental changes in a complex schedule of satellite-tracking antennas and their supporting subsystems. The Deep Space Network (DSN) antennas that it schedules are critical resources, so the DANS system decomposes the problem into a two-level hierarchy, where the scheduling of supporting subsystems is performed in the context of a particular solution to the antenna-scheduling problem. Assignments of antennas to perform specific tasks in fixed time windows are referred to as *tracks*. Given a new activity to schedule, DANS exhaustively searches through the set of constraint-satisfying antenna schedules, which corresponds to a set of tracks, and ranks them as a function of the disruption they cause to the current schedule. Tasks that can be inserted to the current schedule without requiring the cancellation of any existing tracks are treated as if they have zero cost. Where there are conflicts, existing tracks may be preempted based upon a priority-level analysis. More important task are given low priority numbers, and tasks may only preempt tracks of tasks with higher priority numbers. These priority levels appear to be small integers, in the range of roughly 0-9. When conflicts do occur, solution costs are ranked according to the formula

$$\text{Cost} = (\textbf{NAD} * \text{priority}) / (0.9\,\textbf{NAD} + 0.1)$$

Where **NAD** is the number of deletions required to schedule the current activity. When tracks are deleted, the scheduler tries to immediately reinsert them in to the schedule. In addition to this task-at-a-time top-down scheduling, DANS can also be asked to make bottom-up repairs to its schedule in an effort to recover from unanticipated events, such as equipment failure or weather conditions.

We believe that the cost model adopted by DANS can be improved significantly. The priority scheme described in (Chien,et.al, 1997b), allows only very coarse comparisons between solutions. In our work on scheduling systems, we have adopted that view that tasks should be assigned fine-grained values, and the scheduler should be designed in such a way that this information can be used to produce high quality solutions. This approach has several advantages. Determining the demand a job has for resources is more natural and direct than finding the appropriate priority level needed to produce the desired results. It requires no specialized knowledge of the way in which the scheduler interprets priority levels. .In many cases, it amounts to asking a customer how much he is willing to pay. This approach enables us to employ a number of well-known market mechanisms in the search for near-optimal solutions. It allows us to more accurately account for the relative worth of the various tasks, and enables us to apply well-known principles of market economics to scheduling problems.

In this paper, we exploit the advantages of this cost model to develop a set of algorithms for dynamically balancing the computational loads on a set of loosely-coupled processors hosting a community of cooperative software agents. In the special case where there is only one computer, this problem is fairly similar to one which must be addressed by every modern operating system. Conventional solutions to this problem almost always use the sort of coarse-grained scheduling that DANS employs in its top-down scheduler. As a result, it can be very difficult to optimize CPU usage, even when the relative demands of the various processes in the system are known precisely. We examine the case of autonomous agents because it is both fundamental to the work we do at IAI, and because it allows us to consider cases where the migration of tasks between processors is a valid and realistic consideration. While the problem of optimizing CPU usage is not isomorphic to the problem of scheduling satellite arrays, there are a number of similarities. In both cases, there are large numbers of tasks, or consumers, that compete for scarce resources, and the problem is one of finding high levels of global utility through some assignment of resources to consumers. In both cases, some tasks are deemed more valuable than others. Granted, there are fundamental differences in the two problems, and we are not claiming that our results are directly applicable to DNS scheduling. However, we do believe that, for purposes of achieving high-quality assignments of resources to consumers, the fine-grained model of cost and demand has tremendous advantages over coarse-grained systems. In this paper, we show this in the context of managing CPU allocation.

## Computational Resource Allocation and Dynamic Load Balancing

Agents are persistent and goal-oriented, thus they are sensitive to the availability of computational resources. However, the benefit an agent receives from additional CPU cycles varies widely, depending on the importance of the tasks he is working on, the due dates, and also how long he has been working on them (law of marginal returns). Thus the demand for CPU cycles can be different at each agent, and it varies over time. Similarly, the amount of computational resources available on the network can fluctuate as the computers go down/up, or additional workload is added to them. Allocating resources to agents in an optimal way becomes a very intricate problem under these considerations.

Load-balancing and CPU optimization can be considered at three different levels:

- In a single agent community/computer
- Among agent communities on a local area network
- Among local-area networks of agents connected via the Internet.

Within a single community, agents have to time-share the CPU. The problem to decide is what percentage of the CPU cycles each individual agent should get, in order to make the most profitable use of the available CPU cycles. This clearly goes beyond the rudimentary priority-based round-robin scheduling employed by most operating systems.

In a local-area network of agent communities, the problem becomes deciding how to distribute/migrate agents among agent communities in such a way that the aggregate computational capacity on the network is optimally utilized. The optimal solution is at an equilibrium point, where the return for an additional CPU cycle(marginal utility) is the same at each agent community. It is desirable to maintain the equilibrium point with minimal overhead, while being responsive to fluctuations in the capacity of computational resources, as well as the fluctuations in the demands of individual agents. It is also important to take into consideration the cost of migrating agents.

In this paper, we do not consider the third level, which is among local-area networks, connected via the Internet. In the abstract, this is the same problem as load-balancing in a local-area network, with higher costs of communication and agent migration. Thus the same techniques apply. Note that the clustering of load-balancing into several levels makes our approach scaleable to very large systems.

In the next two sections, we will present the technical details of CPU allocation within an agent community, and then the load-balancing protocols across agent communities on a local area network.

## CPU Allocation within an Agent Community

Resource allocation has been a central problem addressed by microeconomic theories. We draw heavily from that body of work. While economists have focused on existence of equilibrium points, our focus is on the computational expedience of finding and maintaining the equilibrium under changing conditions. The time scale makes a big difference: assigning CPU cycles optimally requires responsiveness in the order of seconds, as opposed to in the order of days, or even months in commodity markets. Thus any practical solution must be extremely fast to compute. On the other hand, agents are engineered to have very simple structure compared to people. Hence behavioral assumptions, such as rationality, apply better to agents than to people. As a direct consequence, we expect economic market models to be even more applicable to agents than to people.

In an agent-based application, where each individual agent is a profit maximizing entity, each agent operates by selling his services. In order to provide his services, an agent will need to buy services from other agents. For example, in a factory setting, a particular agent may have the capability to produce a part, but in order to perform this operation he will need to buy input materials as well as time on the proper machine, tool, and fixture, who can perform the operation. In such an environment, it is natural to treat computational resources, just as any other service an agent needs to buy. In making such decisions, the agent needs to decide whether to buy from an expensive, faster machine, to buy now versus at a later time, or choose not to buy at all, if the prospective job is not profitable. The same applies to buying CPU cycles. The agent must decide how much and when to buy/release CPU cycles to maximize his profit.

Let $Q$ denote the computational capacity (cycles/sec) available to an agent community. Assume a mother agent of the community that will be responsible for monitoring the change in available capacity and allocating it among her children agents. Assume the current number of agents in a community to be $n$.

Each agent maintains a demand function, $d_i(q)$, which denotes the rate (in dollars) he is willing to pay for one additional CPU cycle/sec, at his current consumption rate of $q$ cycles/sec. Thus each agent must know how much CPU cycles are worth to him. This is a strong assumption for traditional software; however, agents, which are designed to operate in a market model for profit maximization, would be aware of their computational needs.

A market equilibrium point $<p,q_1,\ldots q_n>$ is defined with the following equations:

$$\Sigma(q_1,\ldots q_n) = Q$$
$$q_i >= 0 \qquad \qquad for \quad i= 1..n$$

$$d_i(q_i) = p \text{ or } [q_i = 0 \text{ and } d_i(0) < p\ ] \qquad for \quad i= 1..n$$

where $p$ is the equilibrium market price. In other words, the CPU cycles are sold to the agents at $p$ dollars per unit. At the equilibrium point, all the available capacity is distributed among the agents. No agent can have a negative share. There is an equilibrium price $p$, such that every agent with a positive share values his next unit CPU cycle at the same price $p$. Thus there is no incentive to swap CPU cycles. The agents whose share is 0 are not willing to buy any cycles at the current market price $p$.

**Theorem 1.** When the demand functions for individual agents are continuous, and strictly decreasing, there exists a unique equilibrium point.

**Synopsis of Proof.** The fact that the demand functions are continuous and strictly decreasing implies that the demand functions are one-to-one and their inverse functions exist. The inverse functions are also continuous, and strictly decreasing. Let $\underline{d}_i(p)$ be the inverse function for $d_i(q_i)$. The function $\Sigma[\underline{d}_i(p)]$ will also be strictly decreasing and continuous, and thus there exists a unique equilibrium point where $\Sigma[\underline{d}_i(p)] = Q$.

An interesting property of the market equilibrium point, is that it is also a Nash-equilibrium point. That is, no agent will benefit from lying about his demand function, when the remaining agents are honest about their demand functions, as stated in the following theorem.

**Theorem 2.** The market equilibrium point is also a Nash-Equilibrium point.

**Synopsis of Proof.** Note that the total amount an agent pays for his CPU cycles is only affected by the quantity he is willing to buy at the market equilibrium price. If the agent has declared a higher demand at that point, then he will be forced to buy additional CPU cycles for a higher price than he is willing to pay. If he has declared a lower demand, then, at the market price, he will miss the chance to buy CPU cycles at a lower rate than he is willing to pay. He loses in either case.

We have presented the equations that determine the market equilibrium point, and proven that there exists a unique solution when the demand functions are constrained to be continuous and strictly decreasing. However, in order for our approach to be practical, we must be able to effectively compute the equilibrium point in the order of milliseconds, in response to fluctuations in the demands from agents, as well as the fluctuation in the available computational capacity.

Obviously, arbitrary demand functions are hard to represent in data structures, and do not lend themselves to closed-form solutions that are efficient to compute. We identify a class of demand functions parameterized with few variables that lead to closed-form solutions which are efficiently computable. This class of functions is also rich enough to represent a wide range of demand functions, and has the following general form:

$$d(q) = a/(b+q); \quad q>=0$$

Depending upon the values the parameters $a$ and $b$ take, this function can assume many different forms. If $b$ is very large, it will appear as a flat line in the operational range. If $b$ is very small, it will appear almost like a vertical line, and in between, it will have a hyperbolic form. The absolute value of the demand can be adjusted with parameter $a$. Furthermore, this leads to a very compact function representation: each agent needs to convey only the values for $a$ and $b$. Note that this function is continuous and strictly decreasing, thus there exists a unique equilibrium solution, which tells us the optimum way to allocate the available computational capacity.

This form also leads to a closed-form solution that is easy to compute:

$$q_i = a/p - b_i$$
$$\Sigma[q_i] = Q$$
$$\Sigma[a_i]/p - \Sigma[b_i] = Q.$$

Let $\Sigma[a_i] = A$ and $\Sigma[b_i] = B$.

$$A/p - B = Q$$
$$p = A/(B+Q)$$

This particular formula does not give the exact equilibrium price: it ignores the constraint that $q >= 0$. A careful examination of the solution reveals that the agents with low demand functions will attempt to operate with negative $q_i$ and sell CPU cycles. If we can tell which agents will have non-zero capacity shares at the equilibrium price, and include only them in the formula, we will have the exact equilibrium price. Fortunately, there is an efficient way to compute the actual equilibrium point, as outlined in the algorithm below:

```
Find Community Equilibrium
1. Sort agents in decreasing order of $a/b$.
2. Let $A = 0$, $B = 0$; $k = 1$; $P = 0$;
3. while ($P < a_k/b_k$) and ($k <= n$)
   {$A = A + a_k$;
    $B = B + b_k$;
    $P = A/(B+Q)$;
    $k = k+1$}
4. for $j = 1$ to $k-1$ do
    $q_j = a_j/P - b_j$;
5. for $j = k$ to $n$ do
    $q_j = 0$;
```

The basic intuition of this algorithm, which computes the exact market equilibrium point is as follows: Initially the set of agents with non-zero capacity allotment is empty. Iteratively add the agent who is willing to pay the highest price for his first CPU cycle to the set, as long as that price (computed by formula $a_i/b_i$) is above the current market equilibrium price. The iteration terminates, when none of the remaining agents are willing to buy any CPU cycles at the current equilibrium price.

The sorting part of the algorithm runs in $O(n \log n)$ time; however, it is needed only once. From then on, updates will take linear time. The iteration in the algorithm is also linear.

Updates occur in response to following events:

- as new agents are added to the community
- as agents in the community are terminated
- as agents change their demand functions
- as the computational capacity available to the community changes.

## CPU Allocation and Load-balancing Across Agent Communities

In this section, we focus on how to best utilize the computational capacity distributed in a local area network. Each agent community in the network has some capacity, and a number of agents. Even though each community is optimizing its CPU cycles using the technique detailed in the previous section, this may not correspond to the global optimum. In a community where demand is high and the capacity is low, the equilibrium price will be very high. In contrast, there may be other communities where the demand is relatively low, and the capacity is large. In such communities the equilibrium price would be low, and the agents in the high-price community can make better use of some of the CPU cycles at the low-price community. Thus we would like to be able to migrate agents from high-price communities to lower priced communities, until all communities have the same equilibrium price. This problem is complicated by the fact that the demand and capacity at each community will be continually changing, due to both external factors (behavior of agents, computers going down, etc.), and internal factors (agents migrating across communities). Agent migration itself has a significant duration, and requires CPU cycles. There is also a tradeoff between the responsiveness of the system, and its stability.

We will first describe the equations that define the theoretical global optimum. This will assume:

- All the information is available at a central location
- There is no latency in the data
- Agent migration is instantaneous

Naturally, these assumptions are not valid in a distributed network environment. We will present techniques that do distributed load-balancing, without these assumptions. When tuned, these techniques will keep the system close to the global optimum, without putting it into oscillation.

One can view a network of agent communities as a single, virtual agent community that houses all the agents in the system, whose computational capacity is the sum of the capacities of individual communities. With this translation into a single virtual community, the allocation techniques developed for a single community can be applied directly to compute the equilibrium price, and the individual share of CPU cycles that each agent will receive. At that point, we have a partitioning problem in our hands: distribute the agents to communities in such a way that the difference between the capacity of a community, and the aggregate capacity share of the agents assigned to that community is minimized. Once the agents are distributed to communities, the equilibrium price at each community, and the share of each agent can be computed. Naturally, there will be some difference in the price level of communities, due to the fragmentation when each agent is assigned to a community. Thus the theoretical equilibrium may not be attainable.

Next, we present a protocol that performs dynamic load balancing. This protocol can handle several important issues:

- New agent communities can appear anywhere in the network
- Existing communities may disappear as their host computers goes down or otherwise become unavailable
- The computational capacity available to a community changes
- New agents are created, terminated
- Existing agents' demand for CPU cycles change

Furthermore, this protocol is parameterized so that it can be tuned to the frequency of changes in the system. In future research, we will address how to dynamically adjust those parameters to changing patterns in the system.

The first step towards moving the system towards the global optimum (equilibrium) point involves estimating the equilibrium price. The theoretical analysis outlined previously requires the information about the capacity of each community and the demand function for each agent in the system. It is not practical to maintain this large body of information at a central location, because it would be severely out of date most of the time. Instead we have devised methods of abstraction to compress this information so that it can be shared among agent communities.

Recall that the market equilibrium algorithm within a single agent community computes two factors: $A = \Sigma[a_i]$; $B = \Sigma[b_i]$. $A$ and $B$ are respectively sums of parameters $a_i$ and $b_i$ for the agents in that community with non-zero capacity share at equilibrium point. Thus $A$ and $B$ contain the summary information on the demand functions of the individual agents in that community.

Let $A_i$ be the sum of $a_i$ of the agents in agent community $i$, and let $B_i$ be the sum of $b_i$ parameters of agents on community $i$. Let $Q_i$ be the computational capacity available to the community $i$. The estimated equilibrium price $P$ is given by the formula:

$$P = \Sigma[A_i])/( \Sigma[B_i] + \Sigma[Q_i] \quad \textit{(equilibrium price formula)}$$

This formula is only an estimate, and not the actual equilibrium price, because it presumes that at the global equilibrium point the set of agents with non-zero share allotment will be the same as in the initial state. Recall that the closed-form solution for the single community equilibrium was inexact, and this was corrected by the Find-Community-Equilibrium Algorithm. We cannot utilize the same technique in the multiple community version, as it would require that the demand functions for each agent be known at a centralized location. However, in most cases, the closed-form solution provides a close approximation, which improves as the system moves towards optimality, and it is exact when the system is operating at market equilibrium. Thus it can be successfully used as a guide for converging to the equilibrium point.

Our load-balancing protocol requires that each mother agent be able to estimate the global equilibrium price. Thus each mother agent will maintain a list containing $<A_i,B_i,Q_i>$ for all communities in the system. We will call this list as the *system load table*. Each mother agent periodically, and in response to significant changes, multicasts her new $<A_i,B_i,Q_i>$ values to the other mother agents, so that this information remains reasonably up to date and accurate. Mother agents also monitor the termination and creation of other agent communities with the same set of messages.

Using the data in the system load table, each mother estimates the global equilibrium price using the above equilibrium price formula. Each mother agent also estimates he equilibrium price at each other agent community using the formula $p_i = A_i/(B_i + Q_i)$. This formula gives accurate answers as long as the parameters are up-to-date. In actuality, it will be a close approximation, because minor fluctuations in those parameters are not reported, in order to minimize message traffic.

Our load-balancing protocol involves two parameters *overload factor* and *underload factor*.

> A community is overloaded if $P_i >$ *overload factor* $* P$, and underloaded if $P_i <$ *underload factor* $* P$.

Where P is the estimated global equilibrium price.

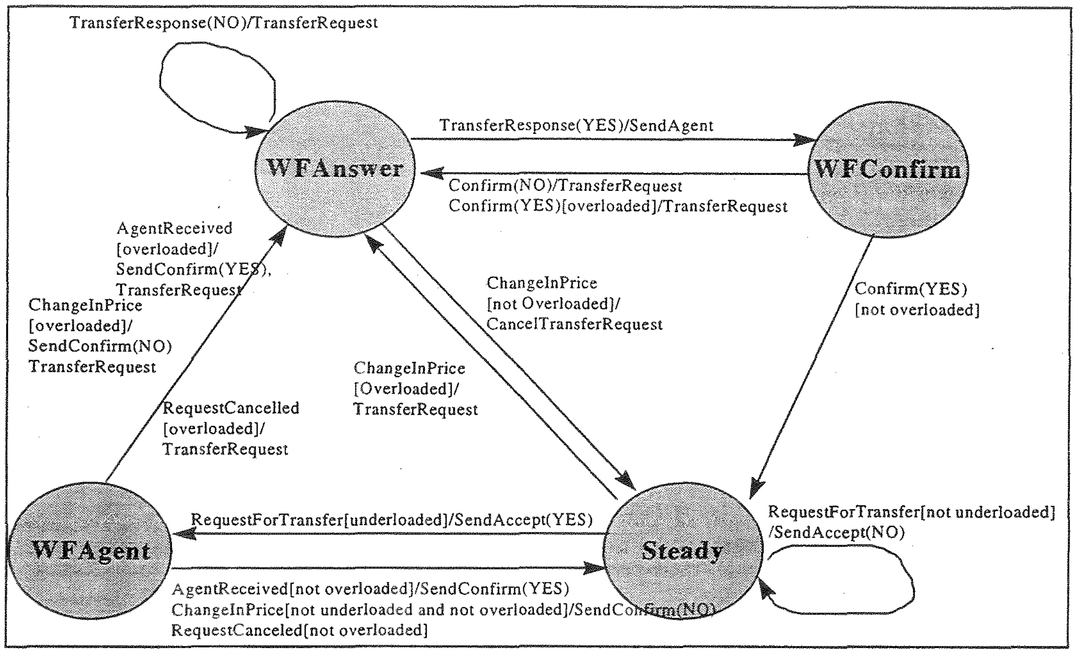
TransferResponse(NO)/TransferRequest
WFAnswer
TransferResponse(YES)/SendAgent
WFConfirm
Confirm(NO)/TransferRequest
Confirm(YES)[overloaded]/TransferRequest
AgentReceived [overloaded]/ SendConfirm(YES), TransferRequest
ChangeInPrice [overloaded]/ SendConfirm(NO) TransferRequest
ChangeInPrice [not Overloaded]/ CancelTransferRequest
Confirm(YES) [not overloaded]
ChangeInPrice [Overloaded]/ TransferRequest
RequestCancelled [overloaded]/ TransferRequest
WFAgent
Steady
RequestForTransfer[underloaded]/SendAccept(YES)
RequestForTransfer[not underloaded] /SendAccept(NO)
AgentReceived[not overloaded]/SendConfirm(YES)
ChangeInPrice[not underloaded and not overloaded]/SendConfirm(NO)
RequestCanceled[not overloaded]

**Figure 1. State-Transition Graph for Load-balancing Protocol**

Overloaded communities attempt to transfer some of their agents to other communities, and a community will accept incoming agents only if the community is underloaded. For values of load factors close to 1, the system attempts to transfer agents eagerly in order to reach close to the optimum point. This is suitable for fairly stable systems with low frequency of fluctuations. For systems with high frequency of fluctuations, it is more appropriate to set the load factors so that the system behaves more conservatively in migrating agents to keep close to the global optimum.

The dialog among mother agents for load balancing proceeds as follows: A mother agent who finds she is overloaded (due to increase in the demand functions of her agents, or a decrease in her computational capacity) probabilistically selects an underloaded mother. The more underloaded a mother is, the likelier she will be selected. The overloaded mother sends a transfer request to the selected mother .If the transfer request is rejected, and the mother is still overloaded, she will probabilistically select another mother. If the transfer request is accepted, she will send one of her agents across and wait for confirmation that the agent is successfully migrated to the other side.

Figure 1 presents the state transition diagram for a mother agent's load-balancing activity. There are a number of short-cuts in the dialog. In particular, mothers can change their minds about sending an agent, or accepting an agent, depending on changes in the market prices.

There are four states:

1. **Steady**: This is the state where the mother agent remains as long as she is not overloaded, and she is not involved in an agent transfer
2. **WFAgent**: This is the state where a mother agent has accepted a new agent and waiting for his arrival.
3. **WFAnswer**: This is the state where the mother agent has asked another mother to take an agent from her, and she is waiting for the response.
4. **WFConfirm**: This is the state where the mother agent has sent one of her agents to another mother, and she is waiting for confirmation that the agent was successfully revived on the other side.

There are also time-out transitions, not displayed in the diagram, so that the protocol can robustly operate in the presence of lost messages, or agent migration failures.

## Conclusion

In cases where the relative demand of processes is known, there is no simple way to reflect this evaluation in the coarse-grained priority, round-robin schedulers found in almost all modern time-sharing operating systems. In this paper, we have shown that it is possible to optimally schedule single processors when consumer priorities are allowed to accurately reflect demand. We have also described a way to find near-optimal assignments of jobs across multiple processors, when jobs are allowed to move between processors. In our opinion, the advantages of fined-grained cost models are indisputable in this sort of complex optimization problems. As described in (Chien, et.al, 1997), DANS falls into this category, but for unstated reasons, the optimization issues are deemed secondary. In many cases, the deficiencies of this approach can be addressed via an intelligent analysis of the domain, and a careful assignment of priorities. However, this can be an arduous task, and it can be simpler to work with a straightforward demand model.

There are other factors that the DANS scheduler must take in to account. For example, schedule stability is highly desirable, and the current system maximizes stability for the highest priority jobs. Market models suggest an alternative approach, in which contract penalties are determined on a task-by-task basis, and encourage stability while still allowing the scheduler some flexibility. These penalties can be adjusted to reflect the desired level of stability.

The top-down approach adopted by DANS is already very time-consuming. When new tasks are being considered, DANS enumerates all of the constraint-satisfying possibilities, and then evaluates the impact of each of these on the current schedule. A finer-grained cost model in this context would greatly complicate the problem, to the extent that the approach would likely be no longer feasible. In our opinion, this is a feature of the top-down search strategy that DANS employs. A number of authors, e.g. (Fisher and Muller, 1995, Sandholm, 1993, and Wellman, 1992) have demonstrated that contract net based, distributed problem solving techniques can be used to achieve high levels of global performance in complēx optimization problems. Our work at IAI (see, for example, Baker,et.al, 1997), is focused almost exclusively in this direction, which can be viewed as bottom-up scheduling and optimization. One advantage of this approach is that it is well-suited to fine-grained models of both cost and demand. In this paper, we have tried to show the utility of such models in a particular context, and suggest that they be considered in the context of satellite array scheduling.

## Acknowledgments

The work described in this paper has been performed by the multiagent research group of Intelligent Automation Inc. Our teaming partners University of Cincinnati, Industrial Technologies Institute, and Flavors Technologies have tremendously contributed to our manufacturing effort. This work has been in part funded by NASA SBIR NAS8-97035, DARPA Agile Manufacturing Program BAA 94-31, Department of Commerce SBIR contract 50-DKNB-6-90112. The views expressed in this paper are solely those of the authors, and do not necessarily reflect the views of the sponsor agencies.

## References

Baker, B., Parunak, V., Erol, K. 1997. Manufacturing over the Internet and into Your Living Room: Perspectives from the AARIA Project. Forthcoming.

Chien, S., Hill, R. W. Jr., Govindjee A., Wang X., Estlin, T., Griesel, M.A., Lam, R., and Fayyad, K.V.1997a. A Hierarchical Architecture for Resource Allocation, Plan Execution, and Revision for Operation of a Network of Communications Antennas. *Proc. 1997 IEEE Int'l Conf. on Robotics and Automation.*

Chien, S., Lam, R., and Quoc V,, 1997b. Resource Scheduling for a Network of Communication Antennas *Proc. Of the IEEE Aerospace Conference,* Aspen, Co.

Fischer, K.,and Müller, H. J. 1995. Cooperative Problem Solving in the Transportation Domain. in Ulrich Derigs (ed.) *Proceedings of the International Conference on Operations Research.*

Sandholm, T. 1993. An Implementation of the Contract Net Protocol Based on Marginal Cost Calculations. *Proceedings of the Eleventh National Conference on Artificial Intelligence.*

Wellman, M. 1992. A General-Equilibrium Approach to Distributed Transportation Planning. *Proceedings of the Tenth National Conference on Artificial Intelligence.*