Distributed Tool Sharing in Flexible Manufacturing Systems

Thomas K. Tsukada, *Member, IEEE,* and Kang G. Shin, *Fellow, IEEE*

***Abstract*— We present a distributed approach to tool management in flexible manufacturing systems (FMS). Tool sharing among cells in FMS enables greater utilization of tool resources, but requires greater coordination of tool use by different cells, especially when tooling requirements change unexpectedly. We discuss how concepts from distributed artificial intelligence (DAI) such as negotiation can provide solution methods. In particular, we propose *polite* methods by which an agent can solve such a problem through negotiation with other agents. We present simulation results for a tool scheduling problem in which polite tool sharing is used for handling unexpected "rush" jobs.**

***Index Terms*—Distributed artificial intelligence (DAI), flexible manufacturing systems, rescheduling, tool scheduling.**

I. Introduction

RESOURCE allocation for a manufacturing system usually occurs within the traditional hierarchical framework. An organizational module receives its allocation of resources from its supervisor on a higher level, and allocates these resources among itself and its subordinates on a lower level. A module having an unexpected requirement for resources unavailable to it will notify its supervisor of the problem. Such unexpected requirements may occur, for example, if new tasks arrive which were not taken into account in the initial allocation of resources, or if machine failure renders some already-allocated resources unavailable. In the past decade, however, there has been increasing interest in new control models for flexible manufacturing systems, emphasizing organizational flexibility, modularity and simplicity of design, and horizontal communication among peers (modules on the same hierarchical level). These models usual emphasize greater autonomy at the level of the *work cell*, the divisions of the shop floor where the work is actually performed. They include Duffie's *heterarchical* model [3], Parunak's YAMS and CASCADE models [8], and more recent work in the distributed AI field (DAI) on multi-agent systems for resource allocation and shop floor control for manufacturing systems [1], [4], [10], [11]. In such systems, distribution of software control modules, and their supporting hardware platforms, parallels the distribution of the manufacturing process itself.

Manuscript received August 6, 1996; revised February 13, 1998. This paper was supported in part by the National Science Foundation under Grants IRI-9209031, DDM-9313222, and IRI-9504412. This paper was recommended for publication by Associate Editor A. Kusiak and Editor P. B. Luh upon evaluation of the reviewers' comments.

The authors are with the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122 USA.

Publisher Item Identifier S 1042-296X(98)03971-8.

Intelligent decentralization of control and peer communication can reduce the degree to which high-level controllers are communication and processing bottlenecks, and typically allows better fault-tolerance, easier modifiability, and exploitation of computational parallelism. While initial allocation of resources may be most efficiently accomplished within the traditional hierarchical framework, relying as it does upon global knowledge of system resource requirements, peer communication may better handle unexpected resource needs which arise during the execution of a manufacturing schedule. For example, an unexpected scheduling conflict between two peer modules may be more quickly resolved by interaction between the two modules, without appealing to and waiting upon the decision of a higher authority.

While peer interaction offers advantages, it often requires more sophisticated coordination, being necessarily less simple than interaction between a supervisor module and its subordinates. A module will have less knowledge about its peers than will the supervisor, and thus may not fully know how its decisions affect its peers, or how a peer may act to resolve a common problem. While Lansky's work on *localized planning* [6] suggests that using only relevant local knowledge makes the search for solutions easier, the lack of global knowledge may preclude obtaining an optimal solution.

This paper explores the application of peer negotiation to the domain of tool management. The goal of tool management is the efficient use of tooling resources, including maximizing tool utilization and allocating tool resources in order to facilitate the processing of manufacturing tasks. In particular, we will consider the problem of sudden unexpected tooling needs, which require a re-allocation of tools, and identify several prototype algorithms and metrics by which to evaluate them. Specifically, we will show that *polite rescheduling*, by which we mean rescheduling of tools by peers in communication with one another using only local knowledge, performs close to good methods using global information, especially when changes to the current schedule are costly. We propose negotiation protocols by which peers which share tools can cooperate to handle unexpected tooling requirements. In this context, we will also consider some basic issues involving distributed approaches to resource allocation and re-allocation.

This paper is organized as follows. Section II provides a formal description of the problem and our approach, and briefly discusses tool management and strategies in FMS. Section III addresses a tool scheduling problem, proposes basic approaches, and presents simulation results which illustrate different issues in resource sharing. Section IV considers

a tool borrowing problem, proposes negotiation protocols, and evaluates these approaches through simulation. Section V presents a summary and conclusions.

II. Tool Management in FMS

Tool management, the allocation and scheduling of tools, is an important problem in FMS. A tool is an implement usually specialized for cutting, drilling, or shaping metal or other matter. It is often separate from the machine using it, so that a tool used at one machine can be removed and transferred for use on another machine. Tools are often expensive, so that strategies for tool allocation and scheduling often have the maximization of tool utilization among their goals.

Tools and machines are used to execute jobs (or tasks). A job may consist of the processing of a single work-piece, or the processing of a batch of similar or identical work-pieces. For example, a job might be to drill a certain number of holes of a specified size, using a specialized drill bit, into each of fifty identical metal parts. The processing time for a job can include the time required for set-up before the actual machining begins, and the time required for the actual machining operation, or repeating operations if the job is a batch. Jobs may have *precedence constraints*, which dictate a partial ordering on the scheduling of jobs. Precedence relations make most scheduling problems more constrained. In this paper, we will not consider the problem of tool scheduling in the presence of job precedence constraints; a similar approach to job-shop rescheduling with precedence constraints in a distributed manufacturing system is presented in [13].

Given a set of jobs, a schedule assigning tools to machines, and jobs to machines, can be constructed. If there were no unexpected changes during the production cycle, schedule execution and tool management during execution would be a straightforward matter. However, unexpected events (or disruptions), such as the arrival of a new job, changes in job priorities, the breakage of a tool, or the breakdown of a machine, may pose problems for schedule execution. The current schedule could of course be discarded, and a new one constructed, taking into account the new requirements imposed by the disruptions. However, this re-scheduling can be costly, not only because of the re-scheduling task itself, but also commitments based upon the original schedule, dealing for example with material transport or personnel, may have to be reorganized. At worst, guarantees made to a customer about delivery times may be violated. Thus, when unexpected events can occur, one goal is to handle disruptions with as little change to existing schedules as possible.

We focus on the problem caused by unexpected tooling requirements. In order to find a solution, some of the constraints of the scheduling problem may have to be relaxed. For example, the processing of a lower priority job may have to be postponed or cancelled. However, negotiation may allow some of the schedule constraints to be relaxed more easily. Tool availability may be less restrictive if a required tool, assigned to another machine, may be borrowed, and we term such an approach *polite*. We examine this type of approach in the context of common tool strategies and a distributed manufacturing model.

A. Common Tool Strategies

Common tool strategies include *mass exchange*, *tool sharing*, and *tool migration* [7]. In the *mass exchange* strategy, each work cell has all the tools required for any task it may ever perform. While this strategy is simple, it is not very efficient if different tools are required for the different tasks one cell can perform; particular tools may often go unused. In the *tool sharing* strategy, each work cell has every tool required for every task it is to perform in the next production cycle. Between production cycles, tools may be moved from one cell to another. Thus, while the management of the tools is somewhat more complicated, tools can be used more efficiently. In the *tool migration* strategy, tools can be moved from cell to cell during the same production cycle (e.g., an eight-hour shift), so that a tool, which has been used but is no longer needed at one cell, can be transported to another cell, where it is needed. Thus, even more efficient use of tools is possible.

Tool sharing and tool migration offer greater flexibility, but are clearly harder to implement. Tool sharing requires information about the locations of the tools and how they will be allocated during the next production cycle. Tool migration requires this information, information about which tools are to be moved from one cell to another, and when the transfer is to take place. *Tool borrowing* is a form of tool migration, in which decisions about the migration of a tool can be made at the work cell level.

B. Distributed Manufacturing System Model

In our model of a distributed manufacturing system, information required for tool management may be distributed. For example, as illustrated in Fig. 1, a shop level supervisory module may have only general information about tooling and job status; a cell controller may have detailed information about cell status and the status of jobs at that cell, while a tool manager module may keep track of tool status (e.g., age, capabilities, etc.) and scheduling. Tool management may require detailed information about cells, jobs, and tools, which may not be centralized in any one module. This model is similar to the contract net-based multi-agent shop-floor control model of Balasubramanian and Norrie [1].

III. A Simple Tool Scheduling Problem

We begin our investigation of using polite re-allocation methods in the tool management domain by considering a very simple tool scheduling problem. Here, simplifying assumptions allow us to consider both optimal and heuristic approaches to the tool re-allocation problem. A more realistic problem (for which optimal solutions are not possible) will be considered in Section IV.

In this problem, we consider the allocation of time slots for use of a tool among tasks which require that tool. We consider a system in which there are tool manager agents and task manager agents. For each tool there is a tool manager agent, which knows the current schedule for the tool (that is, which time slots have been allocated for use by which tasks). For each task there is a task manager agent, which knows the

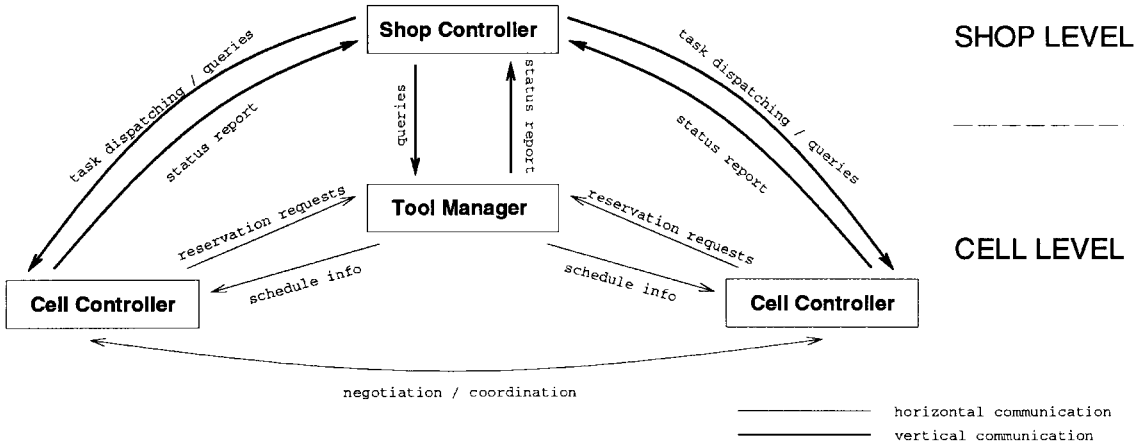

Fig. 1. Distributed manufacturing system model.

tool requirements of the task. Here, a *task* is a generalization of a job; it may be a commonly recurring category of jobs for which tool and processing time requirements are well-known, or it may be a description of a design for which a set of machining operations must be determined.

The constraints for this problem specify, for each task, not only which tools are needed, but also during which time slot windows these tools may be used. This is a simplified version of the time window scheduling problem, in which tasks may be scheduled only during certain time windows. These window constraints may reflect other commitments which may have already been made regarding the task; they are local to the agent which manages the task. A resource capacity constraint requires that only one task may use a given tool during a given time slot. A task may use one or more tools during any given time slot, and to complete processing it must have use of each of its required tools during one of its time slots (not necessarily the same one). The task managers may communicate with tool managers in order to request reservations or scheduling information, and may communicate with one another to coordinate their actions.

A. Problem Description

One obvious goal of scheduling these tasks (allocating tool time slots to tasks) is to maximize the number of tasks which are allowed use of each of their required tools. Given this goal, a backtracking algorithm can be used to find an optimal solution, or some heuristic can be used to find a "good" solution. However, these tasks may arrive individually over time, rather than all at once. Because a schedule represents a commitment to those executing the schedule, and perhaps to a customer, we would like to avoid rescheduling a task once it has already been scheduled. Thus, instead of rescheduling every current task anew each time a new task arrives, our concern is how best to allocate tool use to tasks incrementally. In other words, we will consider ways in which the schedule may be, rather than completely rescheduled from scratch, to accommodate each task arrival.

When a new task arrives, its task manager is responsible for reserving the required time slots on the appropriate tools. If time slots cannot be reserved, then the task must be rejected.

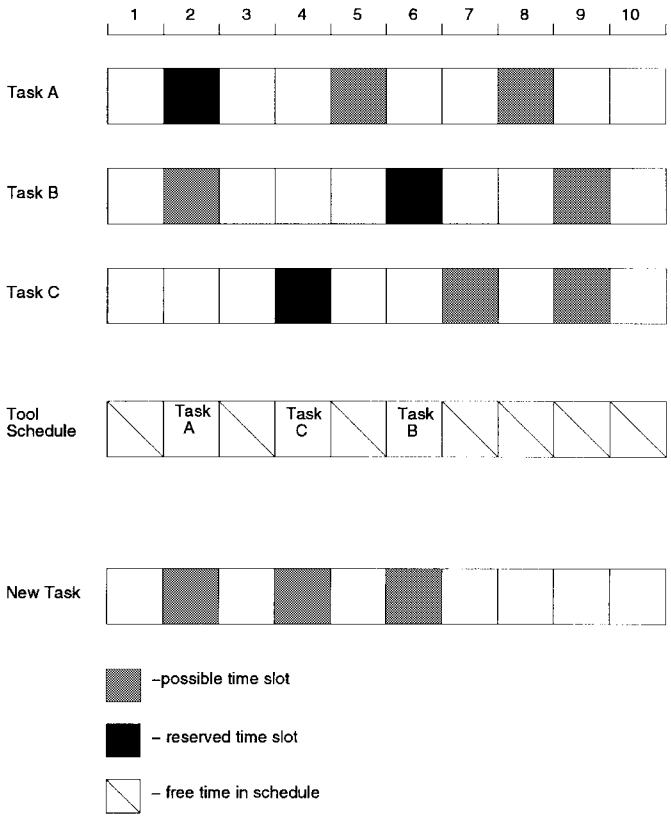

Fig. 2. A simple tool scheduling problem.

Each task requires the use of its required tools for one time slot each. For each required tool, the task has a number of possible time slots during which it may use the tool.

An example of such a tool scheduling problem is shown in Fig. 2 in which three tasks have arrived and have been scheduled. When the new task arrives, either it must be rejected, or one of the already scheduled tasks must be dropped or rescheduled. Fig. 3 shows the original schedule, along with two possible schedules which allow the new task to be scheduled with the previous three. While both schedules include all four tasks, schedule 1 changes the scheduled time for all three of the originally scheduled tasks, while schedule 2 changes only the time for task A, and is therefore less disruptive.

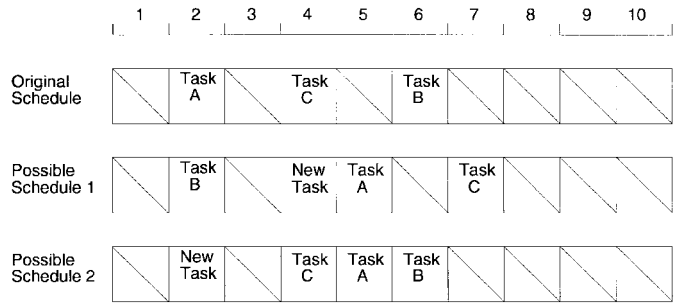

Fig. 3. Possible tool schedules.

As mentioned previously, the responsibility for scheduling a new task belongs to that task's manager. It tries to schedule the task by requesting time slots from the appropriate tool managers. If possible time slots for each tool are available, then the task manager can reserve use of the required tools during appropriate time slots, and the task is thus accepted. It is clear that reserving an available time slot is a nondisrupting action.

However, one or more of the tools may be unavailable during all of the possible time slots. In this case, the environment of the current schedule and inter-agent resource constraints make scheduling the new task in the current schedule impossible. The task manager may handle tool unavailability in one of the following ways. It can decide that its task must be *rejected*, request of the tool manager that the tool be *completely rescheduled*, or *negotiate* with other task managers, to relax local constraints stemming from inter-agent resource constraints. In the traditional bidding approach of the contract net [1], [8], the task would be rejected, as the tool manager would be unable to satisfy the request. When task agents may negotiate with one another, however, a task agent with a tool reservation required by another can become a "contractor," offering use of the tool as if it were a tool manager, if it determines that it can obtain another replacement reservation. We will show that, when complete rescheduling is costly because the collection of global information is not convenient, negotiation to relax local constraints has performance close to optimal in terms of number of tasks accepted, while being less costly than complete rescheduling in terms of the number of tasks which have to be rescheduled.

For example, in the problem in Fig. 2, once the task manager discovers that the task cannot get the tool given the current tool schedule, it can ask the task manager for task A whether task A can use the tool in a different time slot. Because task A can find out from the tool manager that it can also use the tool during time slot 5, its manager would reply to the new task manager that it can change time slots. Thus, the new task can gain use of the tool without changing the plans of tasks B or C. The resulting tool schedule would be the second possible schedule shown in Fig. 3. If task A had not been able to use a different time slot, then the task manager of the new task could have asked the task managers of task B or task C.

1) Problem Statement and Solution Methods: In these simulations, there are s tasks which arrive separately, all requiring use of the same set of m tools. There are s time slots for which tools may be reserved; during one time slot, a tool may be reserved for only one task. Each task has n randomly chosen possible time slots; obtaining use of a required tool for any one of these time slots will satisfy that task's requirement for that tool. In order to simplify the simulation, the first task will not start processing before the last task arrives. As each task arrives, its task manager attempts to schedule it. If it can reserve time slots on each of the required tools, it is accepted. Otherwise, it is rejected. The problem at the ith task arrival is described as follows. Given the time slots, the m tools, $j \leq i-1$ previous successfully scheduled tasks and their tooling requirements and possible time slots, the existing reservations for the j jobs, and the ith job and its possible time slots and tooling requirements, the problem is to find a set of reservations for all $j+1$ jobs by which each job will have reservations for each of its required tools.

We compare two local methods which rely only on local knowledge, a simple scheduling method, and a polite scheduling approach. The goal of each is to schedule as many tasks as possible. In each method, the task manager of a new task first contacts the tool managers to enquire whether any of the possible time slots are available. If an appropriate time slot is available for each tool, then the task is scheduled. If not, in the simple scheduling method, the task is rejected, while in the polite scheduling method, the tool manager informs the task manager which tasks have already reserved those requested time slots. For each unavailable tool, the task manager then contacts the manager of each of those tasks one at a time, to ask whether its tool reservation can be rescheduled. The attempt to reschedule by a task manager avoids the possibility of deadlock, because at any time, only one task manager will be trying to reschedule a given tool. The tool allocation algorithms for a task t requiring one time slot on one tool can be described simply as follows:

n	number of possible time slots for task t;
p_i^t	ith possible time slot for task t;
k_t	number of tools required by t, where $k \leq m$;
Q_t	set of k_t tools required by t, where $Q_t = \{q_1^t, \ldots, q_{k_t}^t\}$;
$\mathrm{sch}_q(j)$	reserve if time slot j for tool q is already reserved, hold if it is tentatively reserved, free otherwise;
T_j^q	task which has reserved time slot j for tool q, if one exists;
r_t^q	time slot reserved for task t on tool q, if any; none otherwise;
h_t^q	time slot tentatively reserved for task t on tool q, if any.

subprocedure *make-tentative-reservation* (t, q)

0. $i := 1$.
1. If $\mathrm{sch}_q(p_i^t) =$ free, then go to 3; else $i := i+1$; If $i \leq n$ then go to 1; else go to 2.
2. Set $h_t^q :=$ none. Report failure. End.
3. Set $\mathrm{sch}_q(p_i^t) :=$ hold and set $h_t := p_i^t$. Report success. End.

subprocedure *tentatively-change-reservation* (t, q)

0. $i := 1$.
1. If $p_i^t \neq r_t$ and $\mathrm{sch}_q(p_i^t) =$ free, then go to 3; else $i := i+1$; if $i \leq n$ then go to 1; else go to 2.
2. Report failure. End.
3. Set $\mathrm{sch}_q(p_i^t) :=$ reserve. Set $h_t^q := p_i^t$ and $r_t^q := p_i^t$; Report success. End.

subprocedure *confirm-reservations*(t)

0. For all $q \in Q_t$,
   0.1. Set $\mathrm{sch}_q(h_t^q) =$ reserve;
   0.2. If $r_t^q \neq h_t^q$, none, set $\mathrm{sch}_q(r_t^q) :=$ free;
   0.3. Set $r_t^q := h_t^q$.
1. End.

subprocedure *cancel-reservations*(t)

0. For all $q \in Q_t$,
   0.1. If $h_t^q \neq$ none, then set $\mathrm{sch}_q(h_t^q) :=$ free;
   0.2. Set $r_t^q :=$ none;
1. End.

procedure *simple-schedule*(t)

0. For all $q \in Q_t$, *make-tentative-reservation*(t, q); If failure, go to 2;
1. *confirm-reservations*(t); report success; End.
2. *cancel-reservations*(t); report failure; End.

procedure *polite-reschedule*(t)

0. For all $q \in Q_t$,
   0.1. *make-tentative-reservation*(t, q); if failure, go to 0.2; else go to 0.5;
   0.2. $i := 1$.
   0.3. Contact task $T_{p_i^t}^q$ and request that it *tentatively-change-reservation*(t, q); if successful, then go to 0.5; else $i := i+1$, if $i \leq n$ then go to 0.3; else go to 2;
   0.4. *make-tentative-reservation*(t, q);
   0.5. Continue;
1. *confirm-reservations*(t); End.
2. *cancel-reservations*(t); End.

We compare these heuristics, which use only local information about who has a reservation for a particular time slot, with optimal methods which use global information about tool schedules to construct schedules with each new task arrival. One optimal method simply maximizes the number of tasks accepted. Another uses a cost factor c for each task that is moved to a different time slot; for each new schedule, this method maximizes $t - cr$, where t is the number of tasks accepted, and r the number of tasks rescheduled. These optimal methods require not only global information, but also a great deal of search, even when bounds are used; the search tree has depth equal to s (the number of task requests), and a branching factor n^m (where m is the number of tools required). (The polite method, because it has much less information to deal with, requires only time on the order of nm.) We will also compare results with a good upper bound, based upon total demand characteristics, for problems which are too big for optimal solutions.

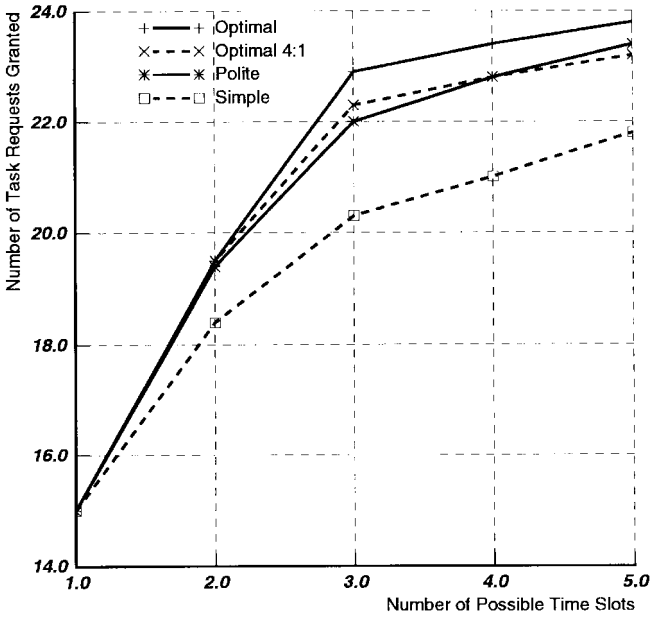

Fig. 4. Simulation results: task scheduling with two tools.

It must be noted that the optimal methods use perfect information about current tool schedules, but not about future task arrivals, or the order in which tasks arrive. If future task arrivals can be perfectly predicted, no task rescheduling would be required, as tasks are only rescheduled to allow scheduling of other tasks. If some information about probabilities of future task characteristics were known, tasks could be scheduled in order to decrease the expected need for future reschedulings, using some robust scheduling method such as that of Drummond *et al*. [2]. The same is true if some prediction can be made about the order in which particular types of tasks arrive. In our simulations, however, in order to address a more general problem, tasks have uniformly random tool-time slot requirements, so no such knowledge about future tasks, or task order, can be used by any method.

2) Simulations and Results: Each of our simulation figures shows the results averaged from 20 simulations. Fig. 4 shows the number of tasks successfully scheduled versus n, the number of possible time slots for each task for each tool. In this case, one tool is required, and 24 task requests arrive, one at a time ($s = 24$). Here, Optimal maximizes the number of tasks accepted, while Optimal 4:1 has $c = \frac{1}{4}$. As would be expected, more tasks are scheduled as n increases and the problem becomes less constrained. The polite scheduling method also schedules about 9% more tasks than the simple method, and comes within 5% of the optimal method. Fig. 5 shows the number of tasks accepted (here with only 16 task requests and time slots) versus the number of tools required, with n set to 3. As the number of required tools is increased, it becomes harder to schedule tasks, as would be expected because the problem is more constrained, and the polite method performance degrades relative to the optimal solution.

While the polite scheduling method is able to schedule more tasks (and thus increase tool utilization), it also imposes additional costs. One of these costs is the rescheduling of certain tasks during the negotiation process. We have men-

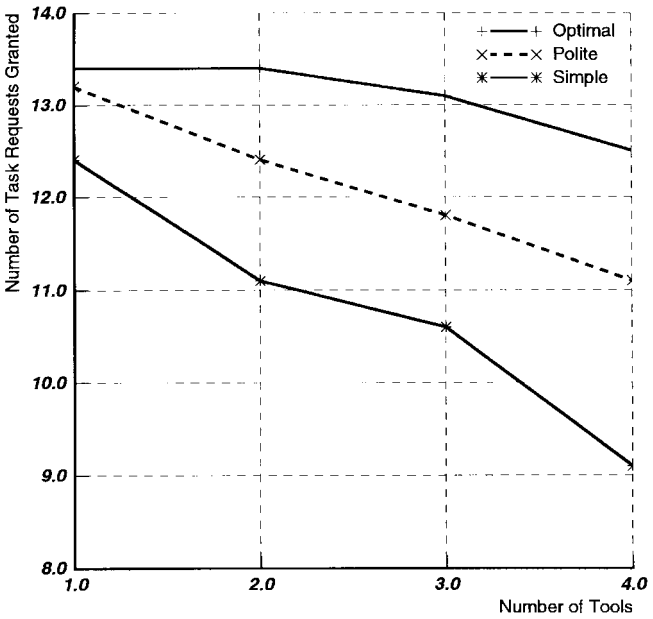

Fig. 5. Simulation results: task scheduling versus number of tools.

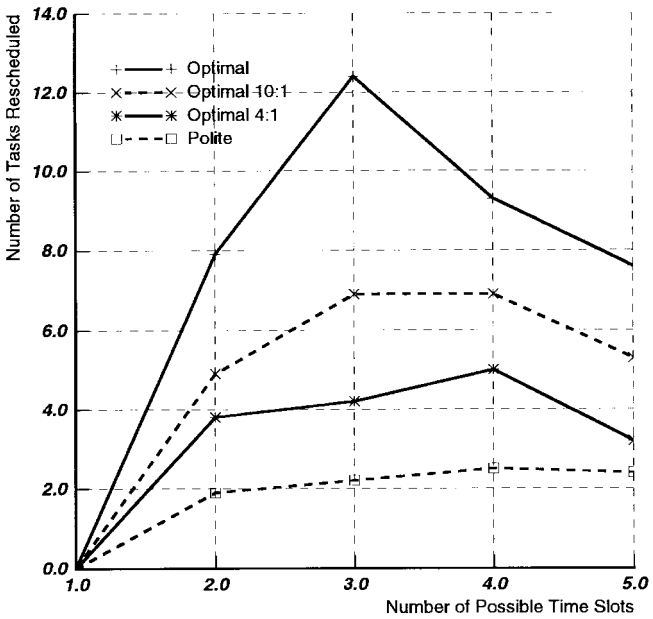

Fig. 6. Simulation results: task rescheduling.

tioned previously that rescheduling is costly, because it often requires changing other commitments associated with the task. Fig. 6 shows the number of tasks rescheduled versus n, where there is one required tools. This number increases initially when n increases, as it becomes easier to reschedule tasks. However, it soon begins to drop, as an increasing n makes the initial scheduling of tasks easier, thus reducing the need for rescheduling. Here, Optimal 10:1 is an optimal method with $c = \frac{1}{10}$, which performed as well as Optimal in terms of number of tasks accepted. The figure clearly shows that the polite method requires fewer task reschedulings than the optimal methods, even when the optimal methods take into account task rescheduling. As c is increased, the optimal method reschedules fewer tasks but accepts fewer tasks, and as c is increased above $\frac{1}{4}$, the task acceptance performance becomes worse than that of the polite method.

3) Simple Negotiation Strategies: Another cost associated with the polite approach is the communication required by negotiation between task managers. While, with increases in network speed, this communication itself might not cause a significant delay, there can be significant costs in network congestion from message traffic, and in computation time from the processing of incoming and outgoing messages. Thus, an important part of negotiation strategies is trying to reduce the number of messages generated.

Two basic ways of reducing message traffic are to contact first those agents most likely to help, and to avoid communicating with agents known to be unhelpful. In order to contact first those agents most likely to help, an agent needs information about which agents are likely to help, and which are not. In this tool scheduling problem, a task with an earliest time slot on a tool is more likely to be able to reschedule than a task with a later time slot, because the later task is more likely to have been unable to schedule earlier and thus is likely to have fewer alternative possible time slots. Sen's work in distributed meeting scheduling [9] shows how search bias determining scheduling preferences can affect communication costs and iterations required for a solution. Here, the same is true when agents are competing for individual time slot reservations, rather than trying to find a common acceptable time slot for a common meeting objective. This early-first preference is equivalent to Sen's *Linear early* search bias, and depends upon agents having common initial scheduling preferences. Similar reasoning can be used if some characteristics about task arrival order is known, though arrival order is random in these examples.

In order to avoid communicating with agents known to be unhelpful, an agent needs information about which agents are not helpful. One possibility is to decide that any agent that was unhelpful in the past will be unhelpful in the future. While this assumption may not hold when agents' situations are frequently changing, in this simple problem, it is a very good assumption. If a task cannot currently change its time slot on a tool, it is unlikely that it will ever be able to do so in the future. Thus, the tool manager for each tool can keep track of which task managers were unable to reschedule on that tool, and inform a requesting task manager only of those other tasks managers which have not yet been unable to reschedule.

Fig. 7 shows the advantages of these methods of reducing message traffic. The graph shows the number of messages between task managers versus n, where two tools are required by each task. In the late-first method, a negotiating task manager contacts other task managers in latest-first order, with regard to the time slot in question. The early-first method does so in earliest-first order. The avoid-unhelpful method does so in earliest-first order, and avoids contact with unhelpful task managers as described above. The figure also shows that the number of messages falls as n is increased, even though, with a greater n, each task manager can negotiate with more task managers; the explanation is that the problem is less constrained and requires less rescheduling with a larger n. Likewise, the advantage of avoiding unhelpful task managers

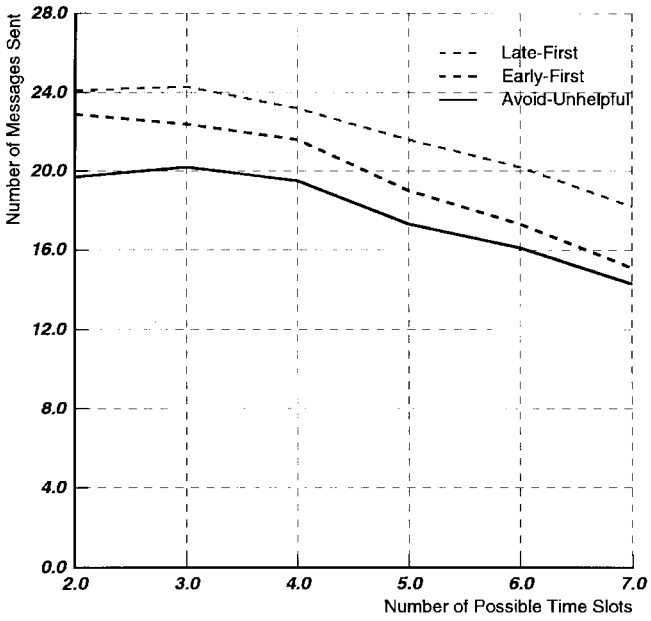

Fig. 7. Simulation results: task rescheduling.

is less with greater n, because fewer task managers are likely to be unhelpful when the problem is less constrained.

B. Discussion

These results show that, in the incremental scheduling of tasks, polite methods using only local knowledge gained through negotiation with a small subset of agents has performance close to that of optimal for maximizing the number of tasks accepted, at lower cost in terms of the number of tasks rescheduled. This is particularly true when inter-agent constraints are limited to the sharing of one tool; as more inter-agent constraints are added with the sharing of more tools, the polite method performance in terms of the number of tasks accepted degrades relative to optimal methods, though the problem size for optimal solutions grows exponentially with the number of tools. This problem, while simple, provides a useful domain in which to study aspects of polite replanning and negotiation as applied to scheduling, particularly because it allows comparison with optimal methods. Our results also provide a basis for our investigation of a more realistic tool allocation and scheduling problem discussed in the next section.

IV. A Tool Borrowing Problem

If a tool is frequently moved among several cells, much potential utilization of the tool will be wasted in transport, given nonnegligible transport times. Tool setup times should similarly be considered, as should the resources required for moving the tool. Thus, some balance should be maintained between moving tools among cells to allow sharing, and keeping the tool in one place to allow efficient utilization.

The *tool sharing* strategy described previously, in which tools are exchanged between (but not during) production cycles, provides this balance. This strategy is similar to using batch scheduling to reduce machine setup costs. In this strategy, a tool is allocated to only one cell during a production cycle (an 8-h shift, for example). The operations at that cell which require use of the tool are scheduled during one such production cycle.

Tool sharing, however, does not address the problem of unexpected tooling requirements, due to a new job arrival, longer-than-expected processing of existing tasks, or machine breakdown. Due to such circumstances, a cell may require a tool during a production cycle during which the tool has been allocated to another cell. Allowing *tool migration* (i.e., exchange of tools during production cycles) during such exceptional circumstances may permit handling of these unexpected events. Using a tool when it is scheduled for use at another cell, however, should not be done haphazardly. Here again negotiation can be useful; the cell which needs the tool can negotiate with the cells which will have the tool, and determine whether and from whom the tool can be obtained.

A. Problem Description

We shall investigate how negotiation can allow useful tool migration, or *tool borrowing*, when unexpected tool requirements arise, but first we should discuss what kind of schedules and assumptions are involved in this problem. For simplicity, we assume here that there are a fixed number of manufacturing cells which share a small set of tools; here we shall consider just one tool. A similar approach should effectively handle situations in which more than one tool is being shared, but will require more complex handling of job sequencing.

Each of these cells has a set of jobs to process. Here, we do not deal with the routing problem in which jobs are assigned to cells. The assumption is that this assignment has been made according to the nonidentical capabilities available at each cell. This job set changes as jobs arrive or are completed. A job may require use of the tool, and can be of high priority (e.g., it is being processed to fulfill a particular order) or of low priority (e.g., it is being processed to build inventory). We assume that the production cycle in this problem is a shift, and that jobs have release times and due times which indicate during which shift a job can begin processing and by which shift a job must complete processing, respectively. Here we assume jobs have no precedence constraints, and each job can thus be scheduled any time between its release time and its due time.

The scheduling goal here is to schedule as many jobs as possible, taking into account the priority levels; scheduling more high-priority jobs is always better. A schedule is constructed for a *scheduling horizon*; shifts beyond the scheduling horizon are not considered in the schedule. Given a scheduling horizon, a schedule is a set of assignments: an assignment of the tool to one cell for each shift within the scheduling horizon, and for each cell, an assignment of its jobs to shifts within the scheduling horizon. In this problem, we are unconcerned how the jobs are sequenced within a shift; we only require that every job assigned to a shift can be processed during that shift. Unassigned jobs in a schedule can be processed after the scheduling horizon, or may be rejected.

We are interested in finding an effective and efficient way to respond to unexpected tooling requirements during schedule

execution. Given a schedule for a particular job set and scheduling horizon, and an unexpected demand for the tool (e.g., an unexpected "rush" priority job which requires the tool), the objective is to meet the new tooling requirement if possible without completely rescheduling all the cells, without requiring use of a centralized scheduling agent, with as little disruption as possible to cell schedules which must be changed, and without having to reject other already scheduled priority jobs.

For constructing the initial schedule, we use a method which first allocates the tool among the cells by considering aggregate potential demand for tool use (as in "texture"-based resource allocation of Fox [5] and Sycara *et al*. [12]), and then assigns jobs to shifts using a knapsack packing algorithm which propagates unassigned jobs forward in the schedule, and which performs close to optimal without exhaustive search. Different weights assigned to jobs in for the knapsack packing problem can be used to represent different priorities among jobs. This initial method can also be used to produce a new schedule in response to unexpected tooling requirements, but it requires a centralized scheduler with global knowledge about tasks and tooling requirements. Here, we do not consider precedence constraints, but were these jobs to have such constraints, jobs with unscheduled successors can be propagated forward along with other unassigned jobs.

B. Handling "Rush" Jobs

In order to investigate how unexpected tooling requirements may be handled, we consider how to handle unexpected "rush" jobs, which arrive at a cell after the schedule has been constructed. Such a job has a due date sometime within the horizon of the schedule; thus, if this new job is to be accepted, either a new schedule must be constructed, or the new job must be fit into the initial schedule. As mentioned previously, making a new schedule from scratch is undesirable. Here we assume that the rush job can only be processed by the cell at which it arrives.

A rush job which requires use of a tool may be harder to fit into the schedule, because tool availability is limited. If there is no tool borrowing, the new job can only be scheduled during shifts before its due date and during which the tool is allocated to the cell. For handling the rush job locally (without tool borrowing), we consider three different methods, among which there is a trade-off between accepting the new job, and modifying the initial schedule as little as possible.

1) LOCAL A: in which the job is scheduled during the latest possible shift in which there is enough idle time to accommodate the new job without rescheduling anything else. It is rejected if no such shift exists.
2) LOCAL B: in which the job is scheduled during the latest possible shift in which there is enough idle time to accommodate the new job without considering the processing time requirements of low priority jobs. Low priority jobs may be rescheduled, but high priority jobs may not be moved.
3) LOCAL C: in which the job is scheduled during the latest possible shift for which rescheduling does not result in any high priority jobs being removed. Low priority jobs may be moved and removed; high priority jobs may be moved.

If tool borrowing is allowed, the cell may ask other cells if it may borrow the tool, if it determines that the job cannot be handled without borrowing the tool. The cell can determine for each shift whether the job can be scheduled in that shift, were the tool available, and then can ask to borrow the tool from the cell which has the tool during that shift. A request to borrow the tool should indicate how long the tool will be needed, including any tool transport time needed. For the cell from which the tool is requested, we consider three possible ways of handling these requests, which are similar to the above local methods, and which have the same tradeoff.

1) BORROW A: in which the tool is lent if the tool idle time during the shift is greater than or equal to the requested time. Thus, lending the tool during this time will not require any rescheduling at the lending cell.
2) BORROW B: in which the tool is lent if the tool idle time, not counting processing time requirements of low priority jobs, is sufficient for the requested time. Low priority jobs may be rescheduled as above; high priority jobs may not be moved.
3) BORROW C: in which the tool is lent if the cell can give up the tool for the requested time, and reschedule without having to remove any high priority jobs. Low priority jobs may be moved and removed; high priority jobs may be moved.

All these "local" and "borrow" methods reschedule or remove jobs using a similar capacity constraint propagation algorithm as the initial scheduling method. The difference is that a multi-capacity knapsack packing algorithm is used to "pack" each shift with tasks, where each tool corresponds to one additional capacity constraint in the knapsack. Like the traditional knapsack packing problem, this multicapacity problem has a psuedo-polynomial time solution of $O(q^{m+1}s)$, where q is the number of discrete time units in a shift, m is the number of tools, and s is the number of jobs. Thus, for a given m, this problem can be solved reasonably quickly.

A final option for handling a rush job is to reschedule all cells completely, using global knowledge of all job requirements, with tool migration allowed between two cells for one shift (as in the tool borrowing case). In this method, all scheduled jobs are unscheduled, the rush job is added to the job set, and the tool is reallocated. Then all possible tool migration between two cells sharing one shift evenly are explored, and the initial capacity constraint propagation method is used to assign jobs to shifts. This does not provide an optimal solution; any meaningful problem in this domain is too large for an optimal solution, but this method does use the initial scheduling method of assigning jobs to shifts, which performs well. This GLOBAL method would not require cooperation per se, but it would require all cells to submit to rescheduling at the same time.

C. Simulations

We performed several simulation experiments to investigate the performance of these methods. For each simulation, a

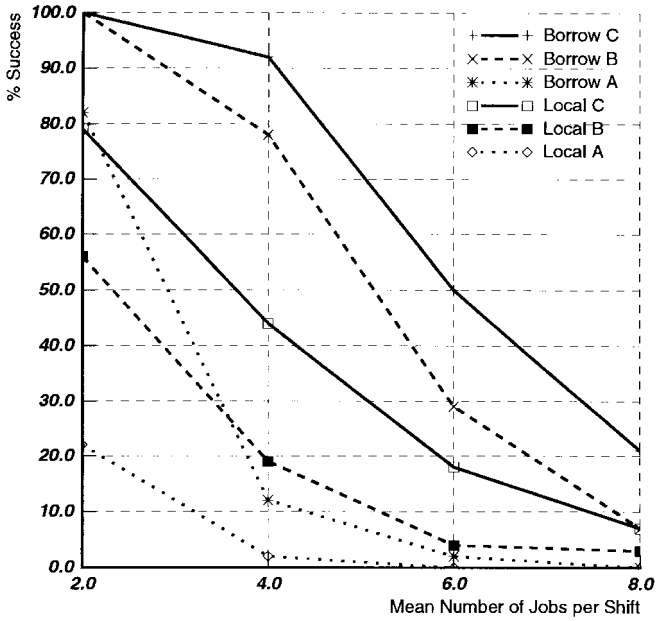

Fig. 8. Rush job handling versus load.

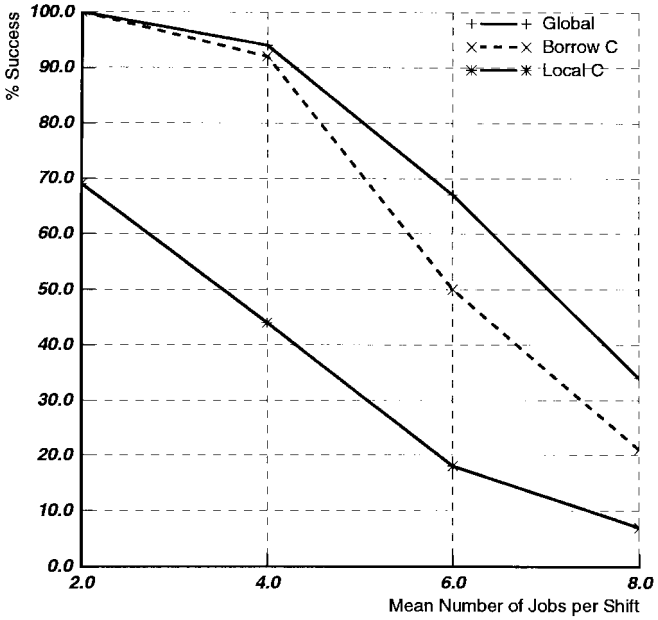

Fig. 9. Rush job handling versus load (continued).

random job set was created using the following parameters: the mean number of jobs per shift per cell (λ), the probability that a job required use of the tool (p), the probability that a job was a priority job, and the mean job processing time. The actual number of jobs for each shift of each cell was a Poisson random variable, which determined the number of jobs for which that shift was the due date; this was not necessarily the number of jobs scheduled at that shift, as a job can be scheduled before its due date. Processing time for each job was a uniform random variable. Each simulation point on the graphs represents the results averaged from 100 job sets. Four separate rush job simulations were run for each job set. One more parameter for rush job simulations was the latency of the rush job, how many shifts in advance it was due.

The simulations were for a system of four cells with a scheduling horizon of eight shifts, and sharing the use of one tool. For each of these simulations, the probability that a job was a priority job was set at 0.5, and the mean job processing time was set at 0.25$\times$ the length of a shift. The time required for tool transport from one cell to another in the middle of a shift was arbitrarily fixed at 0.05$\times$ the length of a shift. If a shift is 8 h, then this tool transport time is 24 min. Unless otherwise specified, each rush job had a latency of three shifts. Where not specified, $\lambda = 4$ and $p = 0.20$. For a system with batch jobs and numerically controlled machines, these numbers are on a realistic level, but are not derived from any one actual manufacturing example.

1) Results: Figs. 8–10, show the performances of the various methods for handling rush jobs. The success axis indicates how often each method was able to schedule the rush job without having to remove any other important jobs. Fig. 8 demonstrates that the requirement not to move any job is very constraining; both LOCAL A and BORROW A, which have this requirement, perform much worse than their less restrictive counterparts. It should also be noted that the flexibility of

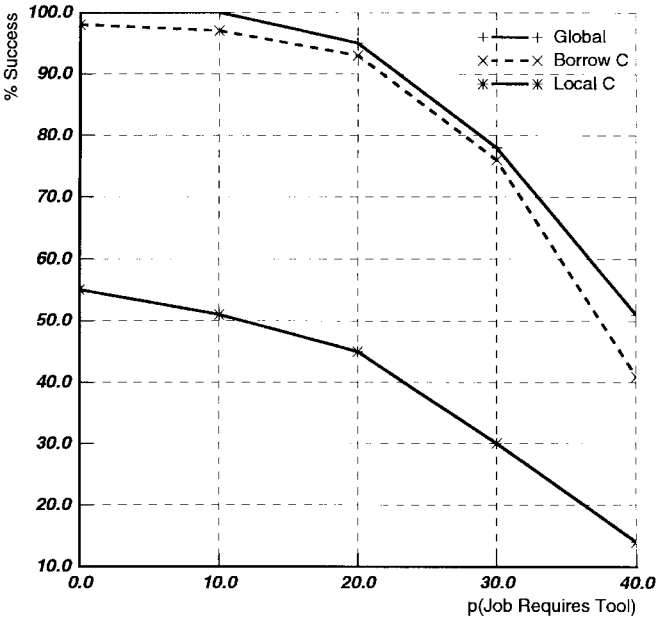

Fig. 10. Rush job handling versus tool requirements.

BORROW C, which is allowed to move important jobs, allows it greater advantage as the problem becomes more constrained.

Figs. 9–11 show the performance of the most effective BORROW and LOCAL methods versus the GLOBAL method which constructs a new schedule using global information. As the problem becomes more constrained, with more jobs per shift or more jobs requiring tool use, the tool borrowing method degrades relative to the GLOBAL method. Nevertheless, when the problem is constrained little enough to allow a good chance of success for the GLOBAL method, the BORROW method using only local information gathered through tool borrowing requests performs close to the GLOBAL method. Fig. 11, showing rush job handling versus rush job latency, also indicates the advantage of tool borrowing

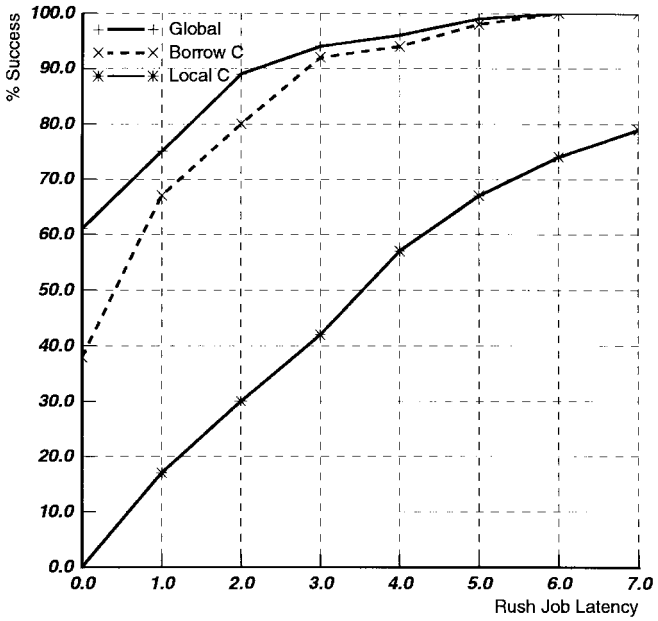

Fig. 11. Rush job handling versus rush job latency.

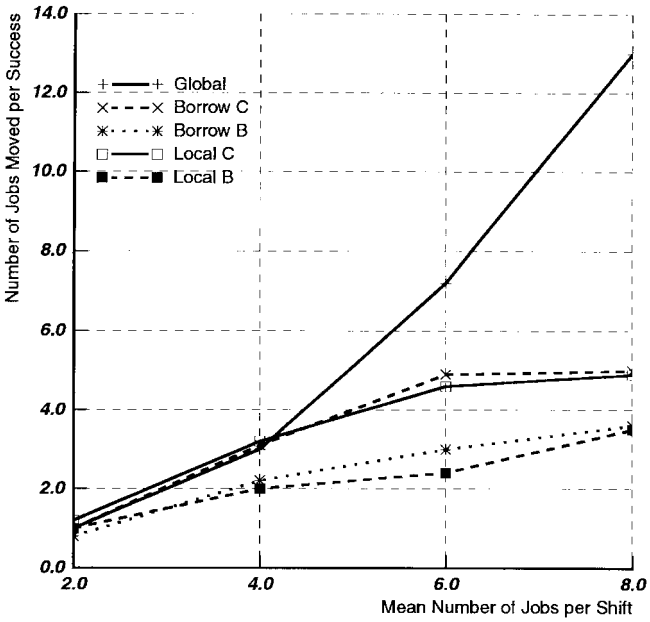

Fig. 12. Jobs moved to handle rush job versus load.

as the problem is less constrained. In all cases, the BORROW method greatly outperforms the LOCAL method.

As in the previous tool scheduling problem, two measures of the cost of handling an unexpected job are the number of jobs which have to be rescheduled in order to accommodate the new job, and the number of message exchanges which are required to borrow the tool. Fig. 12 shows the number of jobs that are moved to different shifts when an urgent job is successfully scheduled. (The LOCAL A and BORROW A methods do not move any jobs.) These results show that, while the flexibility to move important jobs (as in LOCAL C and BORROW C) has a higher cost, the flexibility of tool borrowing does not seem to have a significantly higher cost in terms of rescheduling jobs. LOCAL C and BORROW C move roughly the same number of jobs per rush job acceptance. Thus, tool borrowing itself does not necessarily entail greater disruption, when measured by jobs moved, than local handling methods. Using the GLOBAL method, however, results in many more jobs being moved to different shifts, as might be expected, due to potentially different tool allocations.

Fig. 13 shows the number of message exchanges required by the three tool borrowing algorithms per success, including those exchanges which do not result in successful scheduling of the rush job. The most successful algorithm, BORROW C, also requires the fewest message exchanges, as its flexibility allows both more successful local scheduling, and more successful negotiations. The two other negotiation algorithms require more communication because negotiation with any given cell is less likely to be successful.

Thus, the tradeoff between the local handling and tool borrowing methods is between message exchange (which local methods do not require) and rush job acceptance, while the tradeoff between the tool borrowing methods and the method using global information is between disruptiveness and rush job acceptance. Where current workload is such that rush job acceptance is likely using the GLOBAL method, the most

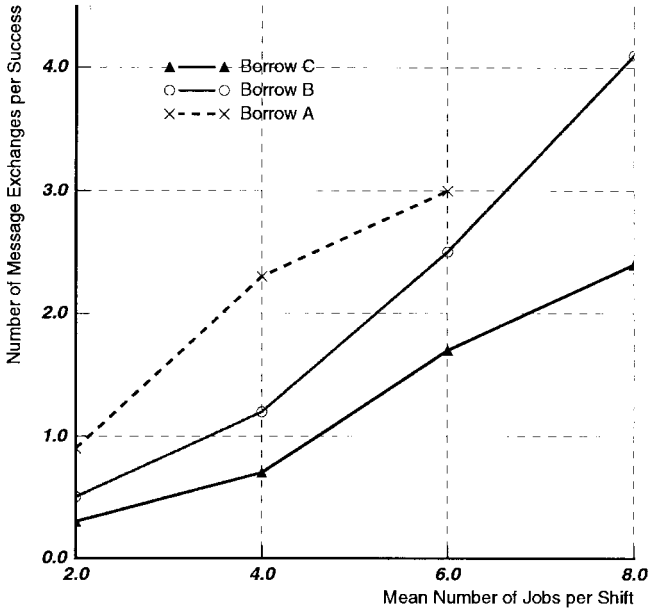

Fig. 13. Number of message exchanges versus job load.

flexible borrowing method accepts the rush job almost as often as the GLOBAL method, moving fewer jobs from their scheduled shifts, and without need for global knowledge. This borrowing method also far outperforms nonborrowing local method, without moving more jobs than its corresponding most flexible nonborrowing local method. Thus, where collection of global information is not convenient, and especially where tool sharing (rather than tool migration) is the desired policy for routine tool scheduling, local rescheduling in response to unexpected tooling requirements is a good method, given the performance and cost measures.

V. Conclusion

In this paper, we have proposed and evaluated tool borrowing protocols, which require an agent to reason about its pri-

orities and capacities when considering a request from another agent to borrow a tool. We have shown that, for unexpected tooling requirements, polite scheduling and rescheduling, using only local information gained through negotiation with a small subset of agents, has performance close to good or even optimal methods using global information, in terms of accepting tasks to be scheduled. Such polite methods also incur smaller costs in terms of the rescheduling of already scheduled tasks. In addition, polite methods show a clear performance advantage over local methods which do not use negotiation. These methods allow many disruptions to be handled locally by one agent, in communication with others, without requiring an appeal to a higher level authority or beginning the resource allocation problem from scratch. Thus, where collection of global information is not convenient, polite scheduling or rescheduling of tools and tasks is a good approach given the performance and cost measures discussed.

We would like, as future work, to integrate the problem of reallocating tools and rescheduling jobs. As mentioned previously, when more than one tool is shared, sequencing of jobs within a shift must be considered when tools can migrate. Likewise, precedence constraints require more sophisticated management of job sequencing. Adding job constraints and allowing multiple tools will necessarily make the reallocation problem harder; however, we are encouraged by the results presented here that this approach to reallocation can be realistically useful.

ACKNOWLEDGMENT

The authors would like to thank S. Birla and D. Ong, General Motors, for their input to the formulation of the tool management problem described in this paper.

REFERENCES

[1] S. Balasubramanian and D. H. Norrie, "A multi-agent intelligent design system integrating manufacturing and shop-floor control," in *Proc. 1st Int. Conf. Multi-Agent Syst.*, 1995, pp. 3–8.
[2] M. Drummond, K. Swanson, and J. Bresina, "Robust scheduling and executing for automatic telescopes," in *Intelligent Scheduling*, M. Zweben and M. S. Fox, Eds. San Mateo, CA: Morgan Kaufmann, 1994, pp. 341–370.
[3] N. A. Duffie *et al*., "Fault-tolerant heterarchical control of heterogeneous manufacturing system entities," *J. Manuf. Syst.*, vol. 7, no. 4, pp. 315–328, 1988.
[4] K. Fischer *et al*., "A model for cooperative transportation scheduling," in *Proc. 1st Int. Conf. Multi-Agent Syst.*, 1995, pp. 109–116.
[5] M. S. Fox *et al*., "Constrained heuristic search," in *Int. Joint Conf. Artificial Intell.*, 1989, pp. 309–315.
[6] A. L. Lansky, "Localized representation and planning," in *Readings in Planning*, J. Allen, J. Hendler, and A. Tate, Eds. San Mateo, CA: Morgan Kaufmann, 1990, pp. 670–674.
[7] W. W. Luggen, *Flexible Manufacturing Cells and Systems* Englewood Cliffs, NJ: Prentice-Hall, 1991.
[8] H. V. D. Parunak, "Distributed artificial intelligence systems," in *Artificial Intelligence Implications for Computer Integrated Manufacturing*, A. Kusiak, Ed. IFS Ltd., 1988, pp. 225–251.
[9] S. Sen and E. H. Durfee, "Unsupervised surrogate agents and search bias change in flexible distributed scheduling," in *Proc. 1st Int. Conf. Multi-Agent Syst.*, 1995, pp. 336–342.
[10] R. Sikora and M. J. Shaw, "Manufacturing information coordination and system integration by a multiagent framework," in *Proc. 13th Int. Distributed AI Workshop*, 1994, pp. 335–362.
[11] K. Sycara *et al*., "An investigation into distributed constraint-directed factory scheduling," in *Proc. IEEE AI Appl.*, 1990, pp. 94–100.
[12] K. Sycara *et al*., "Distributed constrained heuristic search," *IEEE Trans. Syst., Man, Cybern.*, vol. 21, pp. 1446–1461, Nov. 1991.
[13] T. K. Tsukada and K. G. Shin, "PRIAM: Polite rescheduler for intelligent automated manufacturing," *IEEE Trans. Robot. Automat.*, vol. 12, pp. 235–245, Apr. 1996.

Thomas K. Tsukada (S'91–M'96) received the B.A. degree in economics from Haverford College, Haverford, PA, in 1987, the B.S. degree in electrical and computer engineering from the State University of New York at Buffalo in 1989, and the M.S. and Ph.D. degrees in computer science from the University of Michigan, Ann Arbor, in 1991 and 1996, respectively.

He is a Senior Software Engineer at Lockheed Martin Management and Data Systems, Valley Forge, PA. His research interests include scheduling and distributed artificial intelligence.

Kang G. Shin (F'92) received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970, and the M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY, in 1976 and 1978, respectively.

He is Professor and Director of the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Sciences, the University of Michigan, Ann Arbor. From 1978 to 1982, he was on the faculty of Rensselaer Polytechnic Institute, Troy, NY. He has held visiting positions at the U.S. Air Force Flight Dynamics Laboratory, AT&T Bell Laboratories, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, and International Computer Science Institute, Carnegie Mellon University, Pittsburgh, PA. In 1985, he founded the Real-Time Computing Laboratory, where he and his colleagues are investigating various issues related to real-time and fault-tolerant computing. He has authored or co-authored more than 450 technical papers (about 170 of these in archival journals) and numerous book chapters in the areas of distributed real-time computing and control, fault-tolerant computing, computer architecture, robotics and automation, and intelligent manufacturing. He has co-authored the textbook *Real-time Systems* (New York: McGraw-Hill, 1997).

Dr. Shin received the Outstanding IEEE TRANSACTIONS ON AUTOMATIC CONTROL paper award in 1987 for a paper on robot trajectory planning and the Research Excellence Award from the University of Michigan in 1989. He chaired the Computer Science and Engineering Division, University of Michigan, 1991 to 1993. He was the Program Chairman of the 1986 IEEE Real-Time Systems Symposium (RTSS), the General Chairman of the 1987 RTSS, the Guest Editor of the 1987 August special issue of IEEE TRANSACTIONS ON COMPUTERS on Real-Time Systems, a Program Co-Chair for the 1992 International Conference on Parallel Processing, and served numerous technical program committees. He also chaired the IEEE Technical Committee on Real-Time Systems from 1991 to 1993, was a Distinguished Visitor of the Computer Society of the IEEE, an Editor of IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED COMPUTING, and an Area Editor of the *International Journal of Time-Critical Computing Systems.*