Application of Distributed Computing to Batch Plant Design and Scheduling

Sriram Subrahmanyam, Gautham K. Kudva, Matthew H. Bassett, and Joseph F. Pekny
School of Chemical Engineering, Purdue University, West Lafayette, IN 47907

In designing and scheduling batch plants, engineers strive to choose the ones that maximize economic benefits, subject to constraints imposed by process physics and market demands. Even for relatively small design projects and for experienced engineers, the number of practical alternatives is combinatorially explosive. The formalism of mathematical programming provides a convenient way of organizing alternatives and a rigorous justification of choices. Mathematical modeling of these design problems yields large mixed integer linear programs (MILPs). Thus, the effective use of optimization techniques to formalize the design process relies on the ability to solve large models. Distributed computing can be utilized effectively in the solution of MILPs modeling practical design and scheduling problems. This work demonstrates how large MILPs can be solved using a distributed computing platform. The viability of the distributed computing approach is illustrated by presenting results for problems derived from industrial design case studies.

Introduction

Batch and semicontinuous processing have received considerable attention in the chemical industry due to the manufacture of high value added chemicals and drugs. Due to the increase in competition in the chemical industry, it has become necessary to emphasize efficient resource utilization and, therefore, the need for good designs. Optimization techniques are typically employed to obtain such designs. These approaches involve optimizing an objective function subject to physical and operational limitations posed as mathematical constraints. The objective function models the performance criteria dictating the choice of design decisions. Minimizing the capital investment and operating costs of the plant and maximizing the production capacity are examples of widely used performance criteria. The size of problems that can be effectively addressed using this framework is limited by the solution algorithms used, as well as the computing resources available. Solving practical design problems requires the development of algorithms that can handle the large number of constraints and alternatives inherent in these instances.

This article describes an algorithm to address such large design applications using distributed computing. The central idea is to decompose the overall design problem into two stages: the *aggregate design* and the *scheduling levels,* which are similar to the two-level structure involving the materials requirement planning (MRP) problem, proposed by Bitran and Hax (1978). Both stages involve discrete decisions which are represented by integer variables in the mathematical formulation. Such discrete decisions include purchase of equipment in the aggregate design stage and task-unit allocations in the scheduling stage. Thus, these problems are posed as mixed integer linear programs (MILPs) which can be solved in parallel.

Design of Batch Plants

Before the details of the design process and the difficulties associated with it are discussed, a formal description of the design problem would be useful.

Given

- The process recipe structure
- Product demand patterns (with or without uncertainty)
- Process data (yields, processing time, and so on)
- Scheduling constraints (changeover, resource availability, and so on)
- Cost data for equipment purchase and utilization
- Cost data for inventory and utilities (steam, manpower, and so on)

- The horizon of interest (life of the plant)

Find

- The configuration of equipment over time (unit purchase over time)
- Production plan over time (what and when to purchase)
- Detailed schedules to follow the production plans.

Batch processing can be distinguished from continuous processes by the discrete nature of the events that occur in the plant. Since each task performed has a specific processing time, material transfer from one task to another is dictated by the precedence of the tasks and their start and finish times. Material may be transferred without any delay by synchronizing the start time of a task with the completion time of its preceding task. This mode of operation is referred to as *zero wait* (Mauderli and Rippin, 1979). The zero wait mode imposes constraints on the exact sequence and timing of the tasks, and relaxing these constraints may lead to better utilization of resources. Relaxing these constraints implies that management of resources between tasks has to be considered as well. Since there may be a time lag between tasks in the sequence, it becomes necessary to store material between completion time of a task and the start time of its successor task. Placement and sizing of intermediate storage between each set of tasks become an additional design parameter.

Independent of the mode of operation, resource availability has to be considered as well, since it impacts the exact timing of the tasks (scheduling) in the design problem. Thus, scheduling constraints become an inherent part of a general design process in the absence of simplifying assumptions. One such simplifying assumption made in the past is that of cyclic production, where the schedule is obtained for a short time span (typically for a cycle of one batch) and assumed to cycle over the entire horizon. This mode of plant scheduling, known as *campaigning* has received much attention in the last 15 years (Mauderli and Rippin, 1979; Wellons and Reklaitis, 1991; Vaselenak et al., 1987).

Scheduling, however, may not always be important in the design context. In the case of multiple products, if the similarity in product recipes is low, equipment may be dedicated to individual tasks. Scheduling in these cases is simple and does not involve complex mathematical constraints. On the other hand, if product recipes are highly similar, equipment may be shared between tasks (i.e., there may be no unique task-unit assignments), reducing the capital cost of the plant. The disadvantage of sharing is that scheduling becomes important in the plant operation, and models that incorporate scheduling constraints over the entire design horizon lead to extremely large mathematical programs. If dedicated equipment is utilized in the case of processes with high similarity of product recipes, the corresponding design results in high capital cost and underutilization of equipment. For a more detailed analysis of the operating policies and their relation with the recipe network, refer to Reklaitis (1989). The trade-off between dedicated and shared plants has been illustrated by Sahinidis and Grossmann (1991).

The design process also involves decision-making in the light of uncertainty in the predictions of the market demand, supply and processing parameters. In the past (Grossmann and Sargent, 1978; Takamatsu et al., 1973), stochastic parameters have been addressed by representing product demands as continuous variables distributed uniformly within certain bounds (which are determined *a priori*). Uncertainty can be also incorporated in the form of probability distributions. Continuous distributions, however, lead typically to nonlinear expressions in mathematical programs and in most design cases to mixed integer nonlinear programs (MINLP) (Wellons and Reklaitis, 1989). The integer part of design problems is due to the presence of variables representing the purchase of equipment items. Nonlinear problems are extremely difficult to solve, especially in the case of design, where the size of problems is orders of magnitude larger than what can be solved routinely using current MINLP technology. For examples of problems solved using MINLP techniques, see Kocis and Grossmann (1988). Therefore, it is desirable to keep the formulation linear and pose the problem as an MILP. Linearity in the mathematical program can be maintained by introducing the concept of scenarios (Reinhart and Rippin, 1987). Scenarios are a collection of predicted demand levels along with their associated probabilities. The objective of the design is to find an optimal production schedule that takes into account all the scenarios in the decision-making. For details of scenario analysis of market uncertainty, refer to Subrahmanyam et al. (1994b). Shah and Pantelides (1992) considered an MILP formulation with constraints for each scenario, with the objective function being the minimization of capital costs. However, the probability of realization of each scenario is not considered, and all scenarios are treated as being equally likely.

The objective of the design process is to generate a plant configuration which meets certain performance criteria, subject to constraints of the process, the market, and those imposed by the designer. The size of problems that can be effectively addressed using optimization techniques is limited by the algorithms used, as well as the computing resources available. Within the optimization framework, difficulties in the design process are numerous. To obtain optimal solutions, the scheduling constraints have to be handled explicitly in their mathematical form. Unfortunately, the time scale of the scheduling problem which involves conducting basic production activities (batch processing times, clean-out times, and so on) is in the order of hours/days, while the design horizon (involving supply chain considerations, equipment installation frequency, aggregate demand deadlines, and so on) is in the order of months or years. The primary obstacle is that the scheduling constraints yield problems too large to be solved using current solution technology when generated as a mathematical program spanning the design horizon. In addition, the nature of batch processes dictate that there be a task to unit allocation made for every batch execution, which implies in mathematical terms a combinatorial problem (involving discrete decisions). Developing exact algorithms for combinatorial problems is a demanding task. The theory of NP-completeness (Garey and Johnson, 1979) proposes that efforts to develop general-purpose algorithms that can efficiently solve all instances of certain combinatorial problems are likely to be futile. These problems are known as NP-complete problems, and the design and scheduling problems described above belong to this class. However, this theory does not preclude the notion of special purpose solvers which are designed specifically for the efficient solution of problem instances possessing a special structure. Thus, to solve large instances of specific combinatorial problems, the algorithm designer has to invest the time and effort needed to develop special purpose algorithms or make simplifying assumptions to make the mathematical models more tractable. In the past, researchers have usually imposed restrictions on the operation of the plant, thereby mitigating the combinatorial complexity of task-unit assignments over time. As discussed earlier, such restrictions result in suboptimal plant designs.

Approach

A number of approaches have been proposed in the literature to address the design problem. The most rigorous approach involves solving a monolithic model that incorporates the design and uncertainty aspects, as well as scheduling constraints. This framework is comprehensive and an optimal solution to the model is guaranteed to be optimal for the underlying design problem. However, the practical utility of the model is limited by the size of problem instances. The other extreme is to ignore scheduling considerations entirely and adopt a simple operating strategy to make the problem tractable, typically resulting in underutilization of equipment.

Based on the differing time scales involved, the combined design-scheduling model can be naturally decomposed into two stages—one involving aggregate planning (macroscopic) features and the other based on scheduling (microscopic) details. This separation makes solution of the underlying problems more tractable. Separation of the design problem into two stages still implies interactions between the stages, and one cannot be solved without taking into account the influence of the other. The proposed approach is to consider design and scheduling as a two-tier problem with an interactive interface between the two stages. The interface serves to link the two levels and keep them mutually consistent. The solution of the design problem at the top level, called the Design Super Problem (DSP), is used by the interface to set up a series of scheduling problems (Scheduling Sub Stage) taking into account boundary conditions implied by the DSP solution (Subrahmanyam et al., 1994b). The DSP solution therefore serves as a target which the scheduling problems should meet to achieve the predicted goals.

The scheduling aspects of the problem are handled at the Scheduling SubStage. Decisions may be made *a priori* about the mode of operation of the plant, but our approach does not require the assumption of a particular operating mode. For example, it may be necessary to structure plant operations so that production occurs in campaigns of a certain cyclic structure (as discussed before). The mathematical model employed here must be versatile enough to handle the various types of operating modes. Note that this is still a heuristic method and a tie-up with a monolithic model is necessary to establish optimality. The rigorous connection between the monolithic model and our decomposition approach is currently being investigated.

The aim of this approach is to obtain good design solutions without compromising important aspects of the physics of the problem. To this effect, Subrahmanyam et al. (1994b) outlined an iterative decomposition procedure for the design of batch chemical plants. In this work, the inclusion of additional equipment capacity at the lower level to satisfy scheduling constraints is permitted (Subrahmanyam et al., 1994a). This approach is useful in obtaining quickly feasible solutions to large design problems and for obtaining bounds on the overall optimization problem. The proposed method takes advantage of the fact that many plants have to produce products which are of high value and hence, the costs for additional capacity may not have a major effect on the performance function (objective function of the mathematical program).

The first stage in the solution of the design approach is the DSP. We now proceed to outline the mathematical formulation of the DSP. The formulation of the DSP is described in detail by Subrahmanyam et al. (1994b).

Design SuperProblem

The design horizon is divided into a set of aggregate nonuniform time periods, the boundaries of which define the aggregate production deadlines. Each aggregate time period spans several scheduling time intervals and defines a range within which the DSP describes the aggregate production plan (Figure 1). The DSP is derived by summing the scheduling constraints over all the scheduling intervals within the aggregate time period width. In a sense, a surrogation of constraints (scheduling constraints) takes place. For each set of scheduling constraints, a resource limitation constraint (Eq. 2) is generated. Thus, the exact details of events occurring in the design time period are now replaced by an aggregate ac-

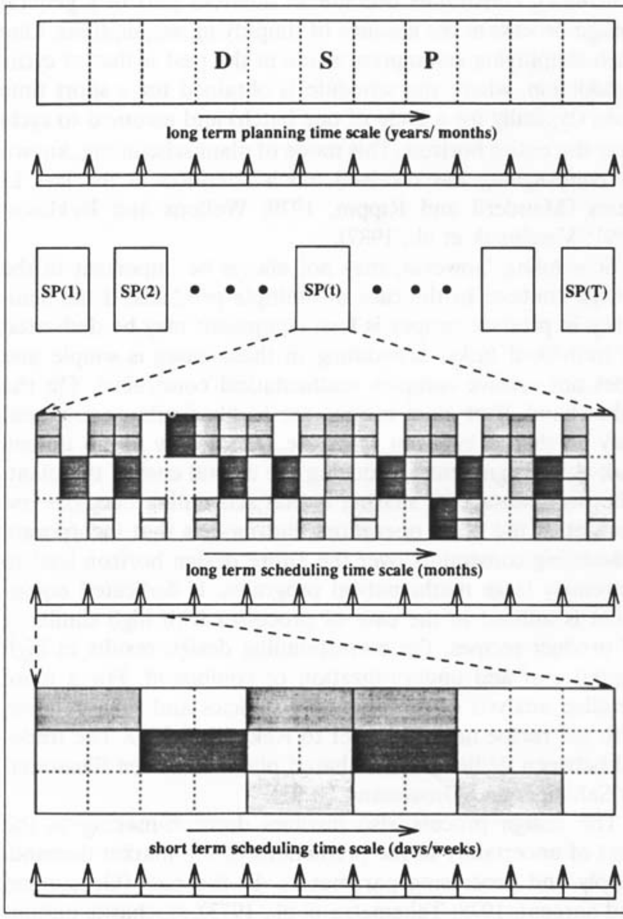
D
S
P
long term planning time scale (years/ months)
SP(1)
SP(2)
SP(t)
SP(T)
long term scheduling time scale (months)
short term scheduling time scale (days/weeks)

Figure 1. Decomposition of time scales.

count. The mathematical formulation of the DSP is given below.

max Expected Net Present Value = (Revenue − Cost)

$$\max ENPV = \sum_{\tau \in T} \left\{ \sum_{k \in K} P_{k\tau} \sum_{s \in S} v^{s}_{sk\tau} q^{s0}_{sk\tau} - \sum_{i \in I} \sum_{j \in I^{\text{equip}}_{i}} C^{ov}_{ij\tau} B_{ij\tau} - \sum_{i \in I} \sum_{j \in I^{\text{equip}}_{i}} C^{of}_{ij\tau} y_{ij\tau} - \sum_{s \in S} v^{b}_{s\tau} q^{b}_{s\tau} - \sum_{j \in J} C^{\text{equip}}_{j\tau} n_{j\tau} - \sum_{j \in J} C^{mp}_{j\tau} N_{j\tau} \right\} \quad (1)$$

$$\sum_{i \in I^{\text{tasks}}_{j}} p_{ij} y_{ij\tau} \le H_{j\tau} N_{j\tau} \quad \forall j, \tau. \quad (2)$$

$$A_{s\tau} = A_{s(\tau-1)} + \sum_{i \in I^{\text{tasks}}_{sp}} \sum_{j \in I^{\text{equip}}_{i}} f^{p}_{si} B_{ij\tau} - \sum_{i \in I^{\text{tasks}}_{sc}} \sum_{j \in I^{\text{equip}}_{i}} f^{c}_{si} B_{ij\tau} - q^{s}_{s\tau} + q^{b}_{s\tau} \quad \forall \text{ resources } s \text{ and } \tau. \quad (3)$$

$$N_{j\tau} = \sum_{\tau'=1}^{\tau} n_{j\tau'} \quad \forall j, \tau. \quad (4)$$

$$q^{s}_{s\tau} = q^{s0}_{sk\tau} + q^{s+}_{sk\tau} \quad \forall s, k, \tau. \quad (5)$$

$$q^{s0}_{sk\tau} \le Q^{s}_{sk\tau} \quad \forall s, k, \tau. \quad (6)$$

$$A_{s\tau} \le A^{\max}_{s\tau} \quad \forall s, \tau. \quad (7)$$

$$B_{ij\tau} \le m_{ij} y_{ij\tau} \quad \forall i, j, \tau. \quad (8)$$

The essential (DSP) formulation is described in brief as follows. Aggregate scheduling constraints (Eq. 2) ensure that any given equipment item is not used for more than the time available during the time period. The material balance constraints (Eq. 3) ensure that material is conserved in the plant by balancing the inflow and outflow of material in each design time period. The other constraints include equipment utility constraints (Eq. 4) to keep track of the number of equipment items available in each design time period, scenario satisfaction constraints (Eqs. 5–6) and gross inventory limitations (Eq. 7) for limiting the inventory storage, batch capacity limitations (Eq. 8) to impose restrictions on batch sizes due to finite unit capacity. The scenario satisfaction constraint set (Eq. 5) represents production in each time period $q^{s}_{s\tau}$ as a sum of two variables $q^{s0}_{sk\tau}$ and $q^{s+}_{sk\tau}$. $q^{s0}_{sk\tau}$ denotes the production amount of material s up to the scenario demand level $Q_{sk\tau}$ according to constraint set (Eq. 6), while anything that is produced above this value is represented by the excess variable $q^{s+}_{sk\tau}$. The amount of material produced should be decided upon after taking into account all the scenarios and the probabilities of their occurrences, and is the same under all the scenarios. To ensure this, $q^{s}_{s\tau}$ is introduced in the constraint set (Eq. 5), and set equal to the sum of $q^{s0}_{sk\tau}$ and $q^{s+}_{sk\tau}$. It is obvious that credit for the sale of a product should only be for the amount produced up to the demand scenario level if that scenario were to be true. If the associated probability that the scenario is true is $P_{k\tau}$, then the credit for the sale of product s is $P_{k\tau} v_{sk\tau} q^{s0}_{sk\tau}$.

The DSP formulation is an MILP with pure integer variables representing the number of equipment items purchased. The performance function against which the design is to be evaluated is the expected net present value of the plant (ENPV). This function is the difference between the expected revenue due to the sale of products and the cost of the raw materials, inventory of materials, and equipment. The sale of products will depend on the production amounts of the products.

Although the DSP is an aggregated form of the scheduling model, a large plant may still translate into a large instance of the DSP formulation. For example, a 41 resource, 31 task network with a horizon of 15 years discretized into 60 time periods of 3 months each yields a formulation instance with over 37,000 variables (total) and 8,000 integer variables. This problem cannot be solved at present to optimality using branch and bound techniques in reasonable time using existing computing resources, unless some efficient special-purpose algorithms are developed. However, even good solutions to these design problems are difficult to generate using a general purpose optimizer, and accelerating the generation of attractive feasible solutions is valuable to the designer.

The solution of the DSP gives the aggregate planning policy and the configuration of equipment over time. The next stage is to evaluate the feasibility of the plant operation by scheduling the production in detail. The following section describes the Scheduling SubStage and the scheduling formulation used.

Scheduling Sub Stage

Each design time period gives rise to a single scheduling problem with a horizon equal to the length of the design time period. Therefore, the number of scheduling problems is equal to the number of design time periods. The scheduling problems are constructed based on the DSP solution—the equipment items implied by the solution are used to schedule the various tasks required to produce the target amounts. The scheduling formulation is based on uniform discretization model (UDM) described by Elkamel (1993). The discretizations are calculated based on the greatest common factor of the processing times and the demand deadlines. Consider the three-month aggregate period for the 31 task process discussed before. The scheduling problems generated thus can still lead to mathematical programs with a large number of variables and constraints.

To make the MILPs generated more tractable, the scheduling problem for each design time period needs to be further decomposed into a number of problems spanning smaller horizons. The axial decomposition strategy described by Elkamel et al. (1993) is used for this purpose. At the scheduling stage, the storage values at the two ends of the horizon are known from the design solution. A problem similar to the DSP can be set up to decompose the scheduling problem into smaller, more amenable problems.

In this article, however, for the purposes of illustrating the effect of distributed computing, the following simpler method is used. The target inventory for the smaller decomposed problems is taken to be a corresponding fraction of the original target storage, the fraction being the ratio of the two horizons. Thus, the storage levels of various materials are decided by linear interpolation. The production amount and target inventory for each material are also obtained by linear interpolation between the initial state and the final storage values of the original horizon. Smaller problems may be scheduled individually and then spliced together to give a feasible schedule for the design time period. The cyclic production assumption has to be noted at this stage. Though the production is to be scheduled in a cyclic fashion, the horizon of interest is still much larger than that considered earlier by other authors, where a cycle is typically equal to a single batch completion time. Also, past work has assumed that such a cyclic pattern would proceed to the end of the horizon, whereas in this case, the cycle terminates at the end of the design time period.

Therefore, there is a cyclic structure locally within the design time period, the cycle time of which is considerably larger than a single batch completion time. The individual problems within each design time period are essentially the same except for the storage values at the ends of the time periods. Production amounts and arrivals are exactly the same in all the smaller problems. Though the storage values are not the same in all the scheduling problems, the usage of each material is equal and the scheduling problems can be considered to be essentially equivalent. Therefore, each small decomposed scheduling problem is to be repeated till the end of the original scheduling horizon. The designer has the degree of freedom to choose the number of smaller horizons into which the original scheduling horizon is to split into. Clearly, there is a limit to the discretization interval of the design time period, as there is a minimum time period required for the production of the products. Scheduling problems thus solved, if feasible, yield a feasible design solution. The solution describes in detail the purchase of equipment, the production plan, and the inventory profiles over time. The following section describes the procedure in the event of a scheduling infeasibility. This procedure identifies bottlenecks in the process and attempts to increase the capacity required to meet the production completely.

Debottlenecking

If infeasibility is encountered in any scheduling problem, it is due to the nonavailability of one or more equipment items or due to overestimation of production capacity. The reason for the infeasibility is that scheduling considerations have been underestimated at the DSP stage. The precedence structure and detailed flow of material are not expressed in the top level formulation and hence the DSP stage has been ambitious in estimating the capacity and in most cases overutilized the available capacity. Infeasibility is not caused by shortage of resources like raw materials or intermediates since material balance constraints imposed in the Design SuperProblem stage hold. Obviously, one (or more) of the unit types is a bottleneck to the process and has to be identified. Identifying bottlenecks in the scheduling problems implied by the DSP solution is an optimization problem. Perturbing the DSP solution by adding a unit of a particular equipment type may yield a better solution in terms of reducing the infeasibility. In this section, we propose an algorithm to identify the bottlenecks in the plant, using the same UDM formulation used for scheduling.

The target production implied by the DSP solution has to be met in every time period to ensure design feasibility. If this production is not met with the current solution, the maximum production is obtained by allowing the target demands to be violated; this relaxation of the original hard constraint (that all the production implied by the DSP solution has to be met) is referred to as *partial demands*. The scheduling problem $SP(t)$, where t is the design time period, is first solved by specifying partial demands on all the products. If all demands are met (satisfy DSP estimated capacity), the schedule is feasible and the algorithm is terminated. The test for feasibility is performed by comparing $G(t)$, which is defined to be the sum of production of the products for the scheduling problem in design time period t, and $G_T(t)$, which is the target production implied by the DSP solution. If $G(t)$ is equal to $G_T(t)$, the production is feasible.

An infeasibility is encountered when the demands are not met in entirety, which is the case where $G(t) < G_T(t)$. To ascertain the bottleneck of the process, a series of problems are generated, each with the addition of one unit type. There will be generated as many problems as there are unit types. These problems are represented by $SP(t, j)$, where j is the unit type added. The linear programming relaxations of these problems are solved, and the cumulative production amount $G^{LP}(t, j)$ compared with $G^*(t)$, the best cumulative production amount discovered so far. If $G^{LP}(t, j)$ is not better than the incumbent $(G^*(t))$, the MILP solution of this problem $SP(t, j)$ is not necessary. This is because the linear programming relaxation is an upper bound for the problem and the solution cannot improve beyond $G^{LP}(t, j)$. If the relaxation yields a better $G^{LP}(t, j)$, the MILP associated with the problem $SP(t, j)$ is solved. The cumulative production amount $G(t, j)$ implied by the MILP solution is compared to $G^*(t)$. A better value of $G(t, j)$ at this stage implies that the addition of unit j increases the production of the products and is maintained as the current best unit to be added, unless a unit k yielding a better $G(t, k)$ is found. The above steps are repeated until the target productivity $G_T(t)$ is found. The algorithm is illustrated in Figure 2.

Parallel algorithm execution

While the two tier framework described above makes the problem more tractable than a monolithic model, the mathematical programming problems generated are still quite large and computationally demanding. For example, the mathematical program representing the DSP for an industrially derived case study contains 2,760 integer variables, 12,085 continuous variables, and 11,058 constraints. The difficulties associated with the large size of these problems is somewhat mitigated by the fact that algorithms to solve them are amenable to parallel execution. The potential reduction in execution times provided by parallel algorithms is important in the context of the design process. Parallel design algorithms can reduce the time required to deliver feasible solutions and thereby allow designers to evaluate a larger number of design alternatives. Computer networks can provide a readily available parallel algorithm execution platform to address such demanding computational tasks. While the use of distributed computing to solve combinatorial optimization problems has not been routine in the past, increased availability of computer networks is likely to make it more prevalent. Lack of software systems that can alleviate the burden

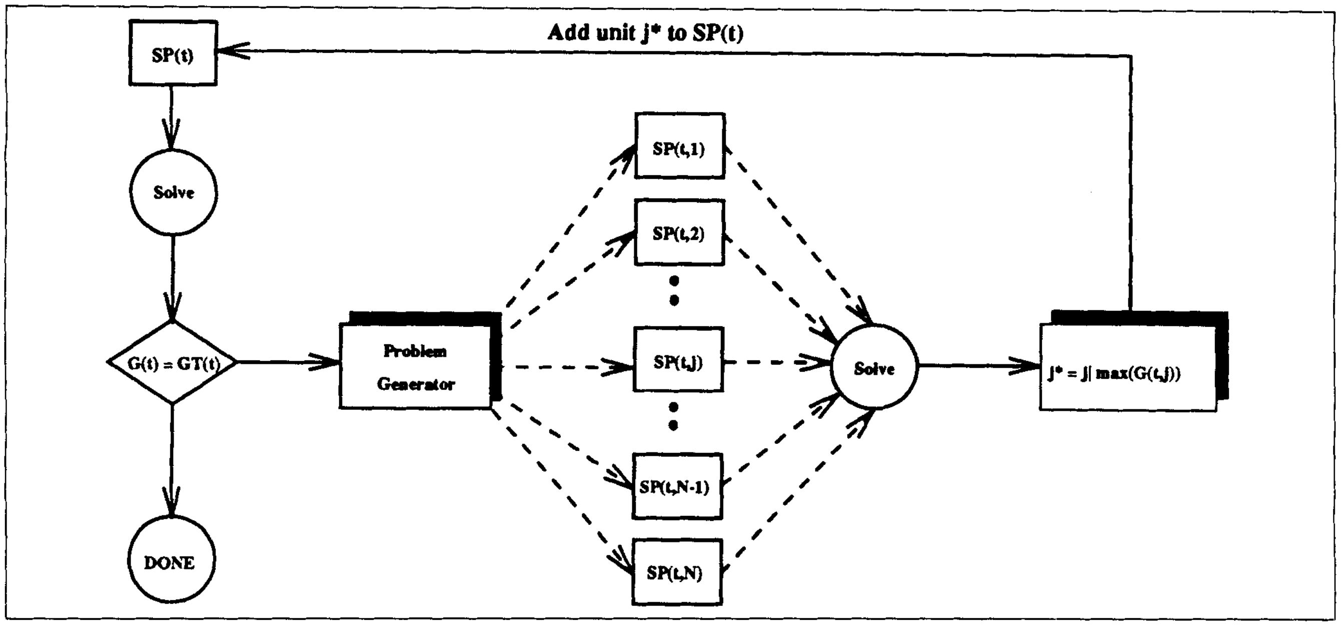

Figure 2. Debottlenecking algorithm.

of developing distributed algorithms is an important factor that has prevented the widespread use of computer networks as a parallel computing platform. This article attempts to demonstrate the utility of distributed computing by describing the development of a distributed algorithm to solve an important class of plant design problems.

The next section discusses the underlying mathematical structure of the above stages. First, the branch-and-bound procedure for the solution of MILPs is discussed. The distributed algorithm that has been implemented to solve the various stages of the design problem is then described.

Branch-and-Bound Framework

The previous section outlined the MILP formulations corresponding to the DSP and the scheduling subproblems. These formulations are solved using the branch-and-bound implicit enumeration method. Branch-and-bound is a well established framework that is at the core of existing methods for rigorously solving hard combinatorial optimization problems (Ibaraki, 1988). In addition, branch-and-bound algorithms lend themselves naturally to parallelization, allowing the use of parallel and/or distributed computing. A formal description of branch-and-bound is provided to clarify succeeding discussions. Consider the following optimization problem P (note that the DSP is posed as a maximization problem, and the following description deals with minimization problems: the min (max) problem can be transformed into the corresponding max (min) form by negating the objective function)

$$Z(P) = \min_{x \in S} v(x) \tag{9}$$

where S is a subset of $\mathbb{R}^n$ and v is a real-valued function known as the objective function. The convention followed is that if an optimization problem of the above form is infeasible $Z(P) = \infty$. The description of the branch-and-bound algorithm follows that of Gendron and Crainic (1993). We assume that P can be solved by enumerating a finite number of points in S and that no polynomial algorithms (in n, the dimension of S) are known to solve it. We also assume that P is infeasible ($S = \emptyset$) or has a finite optimal value. If solving a relaxation of the optimization problem P does not yield a solution $x \in S$, we create K subproblems $P_1, P_2, \ldots, P_K$ with feasible sets $S_1, S_2, \ldots, S_K$ such that $\cup_{i=1}^{K} S_i = S^D \subseteq S$. The optimal solution value for the original problem P and the subproblems P_i are related by the following expression

$$Z(P) = \min_{1 \le i \le K} Z(P_i)$$

If $P_1, P_2, \ldots, P_K$ cannot be solved directly, they can be further decomposed as before. The process of decomposition is known as *branching*. Thus the enumeration scheme proceeds by recursively building a search tree, with the vertices of the tree representing subproblems. Let x^* represent best known solution to P and Z^* be the corresponding objective function value. This is an *upper bound* on the optimal value of P. At the start of the algorithm, Z^* is set to ∞. Whenever a subproblem Q yields a feasible solution to P, Z^* is set to $Z(Q)$ if $Z(Q) < Z^*$. A subproblem Q need not be decomposed if either of the following rules can be applied:

(1) The subproblem has been solved and is either infeasible or generates an optimal solution (i.e., a feasible solution to the original problem P).

(2) $Z(Q) \ge Z^*$.

To apply rule 2, it is sufficient to know a lower bound on the optimal value of Q. The process of generating this bound $Z^l(Q)$ is called the *bounding operation.* Thus if any subproblem Q has a lower bound greater than or equal to a known upper bound on the optimal value of P, it need not be decomposed. Another variant of elimination rule 2, known as the *dominance test* has also been discussed in literature (Ibaraki, 1988). During the construction of the branch-and-bound tree, the subproblems are in one of the following three states: generated, evaluated, or examined. A subproblem is generated, when it is obtained by decomposing another subproblem. When the bounding operation has been applied to a subproblem, it is said to have been evaluated. The subproblem is examined when either the branching operation has been applied to it, or the elimination tests indicate that it is unnecessary to do so—in which case the problem is said to be *eliminated.* Let $\mathcal{N}$ denote the current set of subproblems. A subproblem $P_i \in \mathcal{N}$ is *active* if it is neither evaluated nor examined. Let $\mathcal{A}$ denote the set of active subproblems. A family of branch-and-bound algorithms can now be stated as follows:

(1) Initialization: $\mathcal{A} = \{P\}$, $\mathcal{N} = \{P\}$, $x^* = \emptyset$, $Z^* = \infty$.

(2) Search: If $\mathcal{A} = \emptyset$, stop. x^* is the optimal solution with objective function value Z^*. Else choose a problem P_i from $\mathcal{A}$.

(3) Lower bounding: If $Z^l(P_i)$ is not less than Z^*, discard P_i and go to step 2.

(4) Upper bounding: Determine a feasible solution $x^u \in S_i$. If $v(x^u) < Z^*$, set $x^* = x^u$, $Z^* = v(x^u)$.

(5) Branching: Decompose P_i into $P_{i_1}, P_{i_2}, \ldots, P_{i_K}$ and let

$$\mathcal{A} = \mathcal{A} \cup \{P_{i_1}, P_{i_2}, \ldots, P_{i_K}\} - \{P_i\}$$

$$\mathcal{N} = \mathcal{N} \cup \{P_{i_1}, P_{i_2}, \ldots, P_{i_K}\}$$

Go to step 2. In step 2 of the above algorithm, deciding which subproblem to operate on constitutes the search strategy. In a sequential environment, two search strategies are widely used:

(1) Depth First Search: The problem Q that was last inserted into the list $\mathcal{A}$ is chosen for evaluation.

(2) Best First Search: The problem Q with the smallest lower bound estimate among all problems in $\mathcal{A}$ is chosen. Since the problems in $\mathcal{A}$ are unevaluated, the bounds of their parent problems are used as estimates.

For proof of the finiteness and correctness of the branch-and-bound algorithm, see Ibaraki (1988). In addition to the essential operations of bounding, branching and search, there are a number of other strategies that may be relevant to particular problems. These include upper bounding heuristics, dominance relations, elimination rules and cutting plane techniques. See Ibaraki (1988) and Nemhauser and Wolsey (1988) for details.

Distributed Branch-and-Bound

From the description of branch-and-bound presented in the previous section, it can be seen that the nodes of a branch-and-bound tree can be explored in parallel. Parallel branch-and-bound has been the focus of intense research (see Gendron and Crainic, 1993; Kindervater and Lenstra, 1985, 1988; Beasley, 1987 for a survey). Parallel branch-and-bound algorithms need to address a number of issues including:

(1) *Distribution of Work.* Branch-and-bound nodes need to be transferred among processors exploring a search tree. The distribution has to be effective in the sense that the communication overhead of the parallel algorithm is acceptable.

(2) *Load Balancing.* Work distribution has to be done so that processors make a meaningful contribution to the progress of the tree-search. This implies minimizing the time during which processors are idle as well as ensuring that the processors do not explore unpromising nodes as far as possible.

(3) *Termination.* Determining when a branch-and-bound tree can be terminated. In certain models of distributed computation, this can be a nontrivial issue.

(4) *Information Propagation.* Each branch-and-bound tree has certain global information associated with it. This includes the incumbent solution as well as other information such as the list of variables that can be fixed at certain values in all nodes of the tree, and so on. Such information needs to be propagated to processors in a timely fashion without posing undue burden on the network.

In addition to the above, a flexible framework should allow customization of branch-and-bound components such as bounding functions, branching rules and search strategies. Our research in the field has led to the development of Distributed Control Architecture for Branch and Bound (DCABB), a system for the development of customizable distributed branch-and-bound algorithms (Kudva and Pekny, 1994; Kudva, 1994). DCABB provides the infrastructure needed for distributed algorithms including work distribution and load-balancing. In addition, DCABB provides facilities such as concurrent management of multiple branch-and-bound trees and algorithm specific communication methods. These features make it possible to develop and implement sophisticated strategies that exploit various levels of parallelization opportunities present in an algorithm. The DCABB system has been used to develop a distributed algorithm for the design problem. A detailed description of the algorithm is presented in the next section.

Distributed Design Algorithm

The design algorithm described in the earlier section provides opportunities for varying levels of parallelization. These include:

(1) Parallel solution of the DSP.

(2) Parallel solution of individual Scheduling Problems—here two types of parallelization are possible. Scheduling MILPs associated with different time periods can be solved concurrently; in addition, solution of the individual MILPs themselves can be parallelized.

(3) The LPs associated with the debottlenecking algorithm can be solved concurrently.

(4) The scheduling MILP corresponding to the debottlenecking algorithm can be parallelized.

The efficacy of these strategies is highly dependent on individual problems. Strategies 1 and 4 employ straightforward implementation of parallel branch-and-bound algorithms and strategy 3 is trivially parallel. However, strategy 2 requires a sophisticated algorithm that can coordinate the concurrent distributed solution of multiple MILPs. In addition, the control strategy required to incorporate all four parallelization strategies is nontrivial. We have used the DCABB framework to implement the distributed design algorithm that achieves these goals. We now proceed to describe this algorithm in detail.

Let us denote by N the number of processors used by the distributed algorithm. These processors will be referred to as processor 0, processor 1, ..., processor $N-1$. One of the processors (arbitrarily assigned as processor 0) is used to coordinate the distribution of work among processors and is referred to as the SERVER processor. All other processors are called CLIENTS. Let T represent the number of scheduling time periods in the design problem. The functional description of our distributed algorithm can be summarized as follows:

- Processor 0 generates the Design SuperProblem (DSP).
- The DSP is solved in parallel on all N processors.
- Processor 0 generates Scheduling Sub Problems 0, ..., $T-1$; as each problem is generated it is sent to a free client.
- The scheduling problems are solved concurrently on processors 1, ..., $N-1$. The solution of the MILP is passed back to processor 0. If the number of scheduling problems exceeds the number of available processors, processor 0 maintains the scheduling problem descriptions until a free processor is available to work on the problem.
- When all problems have been dispatched to processors, processor 0 waits for solutions. If the scheduling problem corresponding to a time period is feasible, the design problem is considered to be feasible with respect to that time period. For the design to be considered viable, all scheduling subproblems need to be feasible.
- When the number of processors exceeds the number of active scheduling problems, the free processors are assigned to work on one of the active trees. The active scheduling problem to which an idle processor is assigned is chosen at random.
- When one of the scheduling problems is discovered to be infeasible, the debottlenecking algorithm has to be executed. Let $S=0, \ldots, T-1$ denote the set of all time periods, and $S^{\mathrm{INF}} \subseteq S$ denote the set of time periods whose corresponding scheduling problems are infeasible. Debottlenecking is performed for time period t, which is the earliest time period in the set S^{INF}. The LPs generated by the debottlenecking algorithm (described in the previous section) are solved concurrently on the client processors.
- After adding a unit specified by the debottlenecking algorithm at time t, the scheduling MILP corresponding to time period t needs to be solved. This is done in parallel on processors 0, ..., $N-1$.
- For the modified design problem, all infeasible scheduling problems during time periods $t+1, \ldots, T-1$ have to be solved again as before.

Note that the above algorithm incorporates all four parallelization strategies discussed before. Solving the design MILP and the scheduling MILP corresponding to time period t imply solving one MILP in a distributed fashion using N processors. This is done routinely using the DCABB based distributed MILP solver (Kudva, 1994). Concurrent solution of the scheduling MILPs and the debottlenecking LPs is more involved. The process is best explained by presenting pseudo-code for the algorithms executed by the SERVER and CLIENT processes. The server executes the following algorithm:

```
1. SERVER ALGORITHM
2. {
3.    Generate Design Superproblem (DSP);
4.    Initiate distributed solution of DSP;
5.    Analyze DSP solution;
6.    $t = 0$;
7.    while ($t < T$) {
8.       $t_{\mathrm{bottleneck}}$ = CreateAndSolveSchedulingMILPs
         InParallel ($t, T$);
9.          // solve scheduling problems for time periods $t$
            through $T$
10.      if ($t_{\mathrm{bottleneck}} == T$) break;
11.      while (Scheduling MILP is infeasible) {
12.         SolveBottleneckLPs();
13.         decide appropriate UNIT to add;
14.         initiate Scheduling MILP for period $t_{\mathrm{bottleneck}}$
15.         solve Scheduling MILP;
16.      }
17.      $t = t_{\mathrm{bottleneck}} + 1$;
18.   }
19.   send termination message to clients;
20.   return with feasible solution to DSP;
21. }
22.
23. function CreateAndSolveSchedulingMILPs
    InParallel($t_{\mathrm{init}}, t_{\mathrm{final}}$)
24. {
25.   $t_{\mathrm{bottleneck}} = t_{\mathrm{final}}$;
26.   for ($t = t_{\mathrm{init}}$; $t < t_{\mathrm{final}}$; $t++$) {
27.      CreateSchedulingMILP($t$);
28.      $p$ = FindFreeClient();
29.      if (found $p$)
30.         SendSchedulingMILPToClient($p$);
31.      else
32.         Add MILP to problem queue;
33.   }
34.   while ((not solved all MILPs) AND (not found earli-
      est-infeasible-$t$))
35.   {
36.      wait for a while;
37.      do inter-process I/O and process client messages;
38.      $t_{\mathrm{bottleneck}}$ = FindEarliestInfeasibleTimePeriod();
39.      if ($t_{\mathrm{bottleneck}} < t_{\mathrm{final}}$) break;
40.      $p$ = FindFreeClient();
41.      if (found $p$) {
42.         if (non-empty Scheduling MILP queue)
43.            SendSchedulingMILPToClient($p$);
44.         else
45.            Instruct $p$ to help out working clients;
46.      }
47.   }
48.   return $t_{\mathrm{bottleneck}}$;
49. }
```

The function SolveBottleneckLPs() called on line 12 of the server process algorithm generates a set of LPs as described in the debottlenecking algorithm section and sends them to client processes for solution. Results obtained by solving the LPs are analyzed to determine which unit to add to the design. The algorithm executed by the client processors is as follows:

```
1. CLIENT ALGORITHM
2. {
3.    wait for DSP problem description from server;
4.    work on DSP;
5.    while (not done) {
6.       wait for messages;
7.      while (number of messages ≠ 0) {
8.        if (message-type = = TERMINATION)
9.          done = TRUE;
10.       else if (message-type = = BOTTLENECK-
          MILP)
11.         help to solve bottleneck MILP;
12.       else if (message-type = = BOTTLENECK-LP) {
13.         unpack LP;
14.         solve LP;
15.         send results to server;
16.       }
17.       else if (message-type = = SCHEDULING-
          MILP) {
18.         unpack MILP;
19.         solve MILP;
20.         send results to server;
21.       }
22.       else if (message-type = = HELP-OUT-
          DIRECTIVE) {
23.         poll other clients for work;
24.         help to solve scheduling MILP;
25.       }
26.     }
27.   }
28. }
```

The server directs client processes to help out other clients in solving scheduling MILPs when its queue of scheduling MILPs is exhausted. This is referred to as the *Dynamic Tree Assignment* (DTA) mode. In this case the client that receives the directive polls the other processes, at random, looking for MILPs to solve. When one of the clients, still actively working on an MILP, receives the request from the idle client, it dispatches its MILP and some active tree nodes to the requesting process, thereby enrolling it in the group of processes working on the MILP. The entire algorithm is depicted in Figure 3. The boxes on the left indicate steps in the algorithm while the figures in the right indicate the types of parallelization strategies used for the steps. A simpler implementation would be when the server does not direct client processes to help out in the solution of scheduling MILPs. This is referred to as the *static tree assignment* (STA) mode. The effectiveness of this algorithm is demonstrated in the next section by presenting computational results on problem instances derived from an industrial case study.

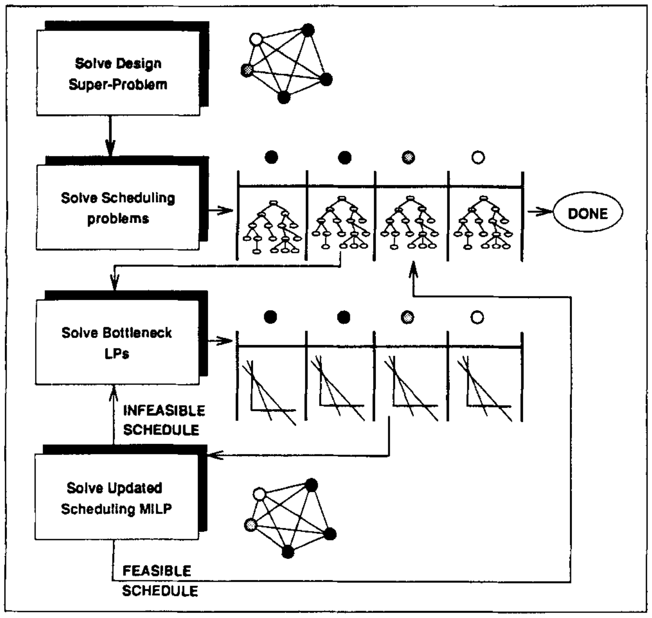
Solve Design Super-Problem
Solve Scheduling problems
DONE
Solve Bottleneck LPs
INFEASIBLE SCHEDULE
Solve Updated Scheduling MILP
FEASIBLE SCHEDULE

Figure 3. Distributed design algorithm.

Computational results

In the previous sections, the test-bed design and scheduling algorithm and the distributed computing aspects were discussed in detail. In this section, several relevant examples to illustrate and validate the algorithm are discussed. The MPS format files for each of these problems are available upon request (email pekny@ecn.purdue.edu).

The goal of parallel and distributed algorithms is to enable faster solution of problems. Hence quantitative measures are needed to determine the effectiveness of a parallel algorithm. In subsequent discussions, we use the generally accepted performance metrics for parallel algorithms—speedup and efficiency. These can be defined as follows

$$\text{speed}-\text{up} = \frac{t_{\text{seq}}}{t_p}$$

$$\text{efficiency} = \frac{t_{\text{seq}}}{p t_p}$$

where t_{seq} is the execution time of the serial algorithm, p is the number of processing elements used in the parallel algorithm and t_p is the execution time of the parallel algorithm. A parallel algorithm needs to demonstrate sufficiently high speedup and efficiency figures to be considered effective. Designers of parallel algorithms strive for linear speedups, i.e., to reduce the execution time linearly with the increase in the number of processors used. However, most practical problems tend to have sequential components that are not amenable to parallel computation, leading to sublinear speedups.

The computing hardware used for all the multiprocessor runs are HP 9000/715 computers, with complete network connectivity. Unless otherwise stated, all single processor runs have been performed on the HP 9000/715 as well.

Distributed Solution of the Design SuperProblem

As discussed earlier, the Design SuperProblem contains aggregate information over the entire life of the plant. The first obstacle in solving the design problem is to obtain quickly and efficiently, good if not optimal solutions to the DSP. Several design problems were solved to optimality using a single processor and tested against multiprocessor runs, the main comparison being the speedup and efficiency obtained through distributed computing. The first three problems (DSP1, DSP2, and DSP3) are based on industrial data for a three-stage process. Within this process, all of the isolated intermediates, dirty solvents, products, and raw materials are storable. The only resources not storable are the mixtures obtained prior to filtration. In DSP1 and DSP2, a total of 13 tasks are to be performed with 22 resources used. Units

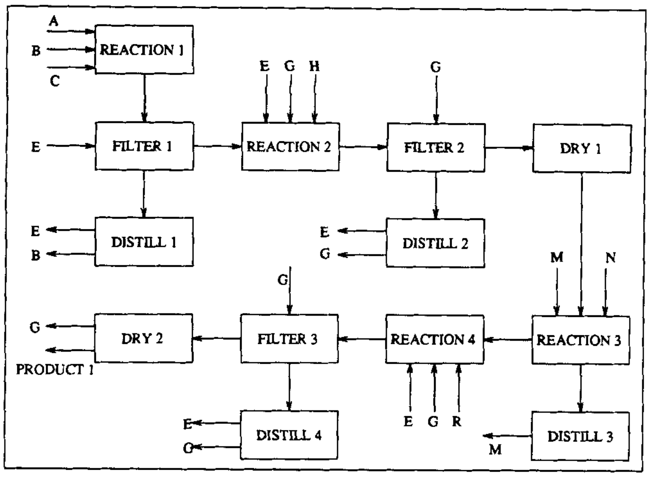

Figure 4. Recipe network for problems DSP1 and DSP2.

should be chosen from nine unit types belonging to five different equipment families (reactor, dryer, centrifuge, filter and filter/dryer combination families). Sale of the single product is during the second quarter of each year during the life of the plant. In addition, the product demand is uncertain with a discrete distribution yielding ten scenarios per quarter. Figure 4 depicting the recipes of problems DSP1 and DSP2 is representative of the process structure of the industrial case study.

DSP1 and DSP2 describe plant design problems spanning 3.5 and 4.5 years respectively with a single product. DSP3 is an extension of these problems with an additional product and spanning 15 years. The process structure is summarized in Table 1. The dimensions of the mathematical program associated with each design problem are given in Table 2.

DSP4 is a problem based on a ten-step process, derived from industrial data. There are 14 resources, three products whose market demands are known *a priori* and units are chosen from five different equipment types. The span of this plant is considerably shorter (1.5 years), and is discretized into nine aggregate time periods. The product demands are such that product P_1 is required only in the odd time periods, while the demand of product P_2 has to be met in every even time period. Product P_3 is required every time period. DSP5 involves the process steps of DSP4 and a demand profile which is uniform over time. The time horizon of interest is extended to 7.5 years and is divided into 30 time periods.

DSP6 differs from the previous problems in structure in that a long chain of steps is not present in its process, and it involves the production of four products in short stages. There are 13 resources, 8 tasks, and 14 time periods in this process (Table 1). The choice of unit types over which design decisions have to be made is limited to four. The essential difference in this problem is that it is a typical example of a short-chain process and lies at the other end of the processing spectrum, while all the other previously discussed problems involve long-chain processes.

Table 1. Process Structure Data for the Design Cases

Name	No. Units Types	No. Tasks	No. Res.	No. Prod.	No. Time Periods	Horizon (yr)
DSP1	9	13	22	1	14	2.5
DSP2	9	13	22	1	18	3.5
DSP3	6	27	41	2	60	15
DSP4	5	10	14	3	9	1.5
DSP5	5	10	14	3	30	7.5
DSP6	4	8	13	4	14	1.2

Table 2. Distributed Computing Results for the Design SuperProblem

Name	Variable	Integer	Constraint	No. Procs.	Soln. Time	Bound Gap (%)
DSP1	1,712	126	1,370	1	555	5.285
				8	104	5.285
DSP2	2,193	162	1,760	1	13,319	0.088
				8	1,395	0.049
DSP3	14,845	2,760	11,058	1	20,389	8.415
				8	2,608	8.36
DSP4	590	45	495	1	1,077	0
				6	334	0
DSP5	2,024	150	1,680	1	27,510	6.95
				10	1,655	3.94
DSP6	728	56	462	1	724	0
				6	362	0

An attempt to solve DSP1, DSP2, and DSP3 was made using the CPLEX-MIP (CPLEX, 1993) solver. CPLEX-MIP was unable to find a solution for DSP1 in 60 min using an HP 9000/755 computer. DSP2 was solved to the first feasible solution which was only within 30% of the optimal solution in under 2 h of time using the same computer. In the case of DSP3, CPLEX-MIP was unable to find a solution after 8 h. The utility of general purpose solvers in solving large optimization problems is limited, as is apparent from the above experiments.

Within the DCABB framework, the CPLEX LP solver (CPLEX, 1993) was used to solve the linear programming relaxations corresponding to the branch-and-bound nodes. The Design SuperProblems of several design cases were solved using single processor (serial run) and multiple processors (distributed run). The structure of the problem can be exploited at this stage by specialized rounding techniques leading to good feasible solutions. These techniques can be applied at any point of the branch and bound tree in an attempt to obtain good solutions and reduce the search space. In problems DSP[1–3], staged rounding (Subrahmanyam et al., 1994b) was applied at each node in an attempt to obtain feasible solutions quickly.

The results of the distributed runs are listed in Table 2. The last column of the table represents the percentage difference between the incumbent solution and the best known lower bound at the time of termination of the run. This is referred to as the bound gap. Note that when a problem is solved to optimality, the bound gap is zero. In all cases considered, there is considerable speedup in the computational effort involved in solving the optimization problems. However, in many instances, a proportional speedup corresponding to the effective number of processors is not observed. For a detailed discussion of this phenomenon refer to Kudva (1994). Briefly, this is due to the fact that a parallel branch-and-bound search has to explore multiple nodes concurrently, leading to a different execution profile than the serial search. Potentially, the parallel search may explore parts of the branch-and-bound tree that would not be explored by the sequential search leading to sublinear speedups. On the other

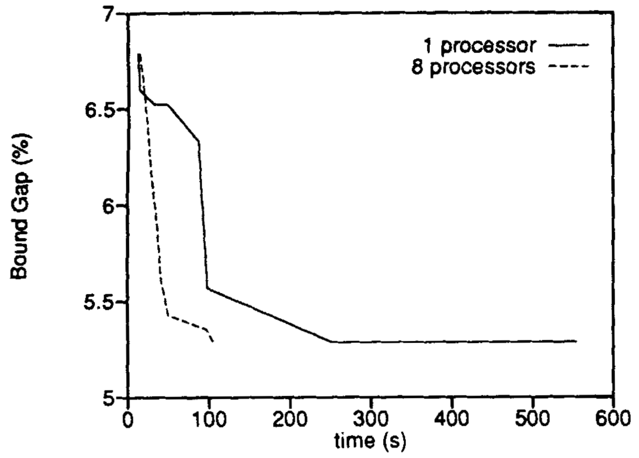

Figure 5. Progress of bound gap for problem DSP1.

hand, due to the different execution profile, the parallel search may discover incumbents quickly, leading to a fathoming of a large number of unexplored nodes. Thus it is possible to achieve superlinear speedups. Tracking the bounds over time will better illustrate the effectiveness of distributed branch-and-bound search. The bound gap tracking over time is illustrated for problem DSP1, DSP2, and DSP3 is shown in Figures 5, 6, and 7, respectively.

Figure 5 shows a comparison between the serial and distributed runs for DSP1. Neither method was able to close the bound gap under 5% of the optimal solution value. However, using 8 HP 9000/715 processors enhanced the solution time to 104 s, when compared to 555 s for the serial case. Figure 6 compares the bound gaps for problem DSP2. The bound gap is almost closed (to within 0.1% of the optimal), with the distributed solution being obtained 9.5 times faster than the serial one (the solution times for the distributed and serial runs are 1,395 and 13,319 s, respectively). CPLEX-MIP, for this case, had only obtained a solution which was within 30% of the optimal. Problem DSP3 proved considerably more difficult to solve, with both methods achieving some bound gap closure, as shown in Figure 7 (8.35% for the distributed case in 2,608 s and 8.4% for the serial run in 20,389 s), with the distributed algorithm converging to a better incumbent than the sequential run.

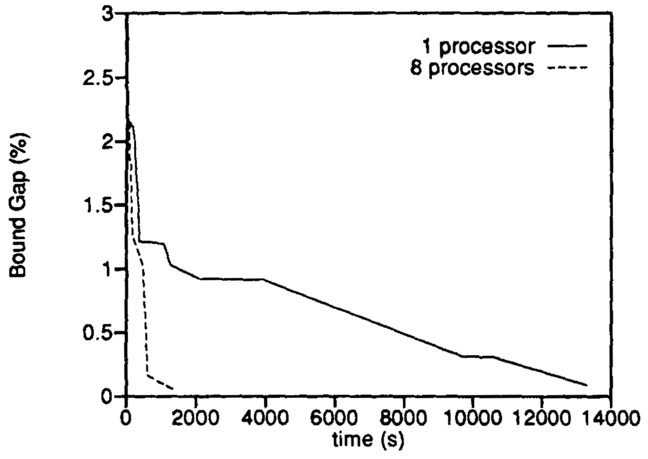

Figure 6. Progress of bound gap for problem DSP2.

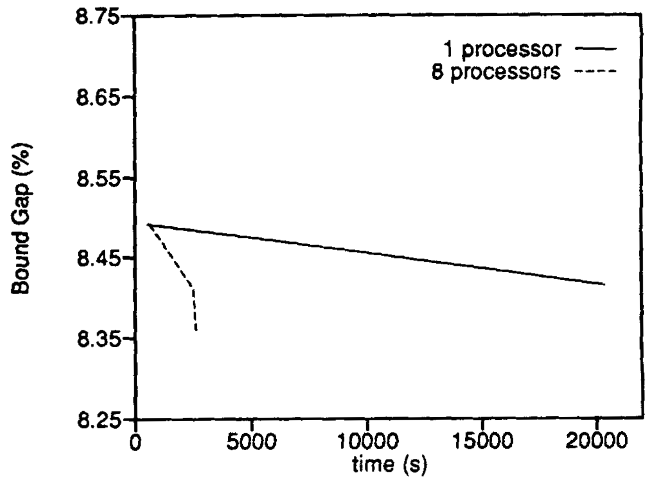

Figure 7. Progress of bound gap for problem DSP3.

DSP4 is solved to optimality by both the serial and distributed runs. Note that the DSP formulation of this problem is considerably smaller than the previous three that were considered. DSP5 is, as mentioned earlier, an extension of DSP4 and is therefore a much more computationally intensive problem to solve. The distributed case yields a better solution than the serial run, with the bound gaps being 3.94% and 6.95%, respectively. The success of the distributed run in this case is also exhibited by comparing the run times, with the distributed run utilizing 1,655 s to obtain its best solution, while the single processor takes over 20,000 s without reducing its bound gap. DSP6 is solved to optimality by both runs, and this may once again be attributed to the size of the problem (in particular, the number of integer variables). The inference from the above results is that distributed computing performs extremely well for larger instances of the Design SuperProblem, with the efficiency decreasing as the problem size reduces. Employing multiple processors to solve small problems may prove wasteful, since the search space is small, and the optimal solution is found quickly even with a single processor.

Distributed Solution of the Overall Design Problem

Once the DSP has been solved for a good feasible solution (or optimal solution), processor 0 takes charge of setting up scheduling problems for each aggregate time period of the DSP. Clearly, the scheduling problems are independent at this stage, and parallel processing of the problems would be natural. As processor 0 creates scheduling problems, it sends it to any free processor, which starts working on the MILP. After it dispatches all the problems, processor 0 acts only as a monitoring device, waiting for the problems to return after solution. Each processor works on its own assigned problem, thus acting in a single processor mode.

Ideally, a speedup proportional to the number of processors is expected from this algorithm (taking only the scheduling computations into account); unfortunately, this is rarely true in practice. The following analysis justifies this observation. Let the time taken for problem i to be solved be represented by t_i. Then, for the serial run, the time taken for the termination of the scheduling algorithm is $\Sigma_{i=0}^{N-1} t_i$. For the distributed run, assuming there are an equal number of processors as there are problems ($N_p = N$), let problem i be assigned to processor i. The time taken for the termination to solve all the scheduling problems would be $\max_i\{t_i\}$.

Then, the following relationship is valid

$$\sum_{i=0}^{N-1} t_i \leq N * \max_i \{t_i\} \tag{10}$$

The time taken for the serial run $\Sigma_{i=0}^{N-1} t_i$ is always less or equal to the term $N \max_i\{t_i\}$, which is the product of the number of processors and the distributed computing time (maximum of the individual scheduling problem solution times). The only scenario where the equality would hold is when $t_i = t_{\max} \forall i$. This is true only when each scheduling problem has the same solution time.

The speedup is defined as the ratio of the time taken for the serial run to that of the distributed run.

$$S \equiv \frac{\sum_{i=0}^{N-1} t_i}{\max_i \{t_i\}} \tag{11}$$

From this definition and the previous relation (Eq. 10), it is easily seen that

$$S = \left(\frac{\sum_{i=0}^{N-1} t_i}{\max_i \{t_i\}} \right) \leq N \tag{12}$$

The only scenario, as discussed before, where the speedup S would be equal to N is when $t_i = t_{\max} \forall i$. In a branch and bound procedure, it would be extremely difficult to guarantee that each scheduling problem would be solved in exactly the same time as every other. There is always disparity in the solution times, due to the difference in the size of the problems, or due to the nature of the solution space and, hence, the difficulty in solving them.

Problems DP[1–5] are all derived from the same industrial case study as problems DSP[1–5]. The essential difference in all the case studies is to observe plant design response to varying demand data. Though there is no drastic change in the process structure, changing the demand patterns greatly affects the final design(s). This is evident from the results shown in Table 3. Column 4 (No. SP) indicates the number of design time periods and hence the number of scheduling problems implied by the DSP. Columns 5, 6 and 7 (SP time, DBP time and overall time) indicate the wall-clock time required to solve the DSP, the scheduling MILPs, and the debottlenecking algorithm respectively. The total execution time is given in the last column. All runs were made using HP 9000/715 processors. Since this section deals with the solution of the design problems (including the scheduling problems), the results in Table 3 show the solution of multiple MILPs and it would be meaningless to consider the progression of the bounds. However, comparing the speedup obtained for the DSP and the scheduling problems would be useful.

Table 3. Distributed Computing Results for Design Case Studies

Name	No. Procs.	DSP Time	No. SP	SP Time	DBP Time	Overall Time
DP1	1	166	9	794	0	962
	6	157	9	281	0	442
DP2.1	1	1,077	9	1,205	200	2,534
	6	334	9	478	99	980
DP2.2	1	949	9	—	—	—
	6	362	9	1,996	134	2,615
DP3	1	129	9	552	68	777
	6	113	9	179	45	372
DP4	1	13,860	18	—	—	—
	6	7,547	18	427	51	8,064
	10	2,730	18	344	44	3,164
DP5	1	489	9	3,065	407	4,085
	6	286	9	2,320	200	2,926

The scheduling horizons associated with problems DP[1–5] span 3 months, or in terms of the discretization intervals, over 1,500 intervals. A mathematical program of that size would be extremely difficult to solve. Hence, the procedure outlined earlier (in the discussion on the Scheduling SubStage) is used to decompose the problems into more amenable scheduling problems. A large horizon scheduling problem is treated as a set of simpler schedules cycling over the larger horizon. Therefore, a 1,500 h schedule is composed of a 100 h schedule cycling over the 3 month period.

Due to its small size, problem DP1 exhibits no speedup in the DSP solution time. An optimal solution is discovered in less than 200 s, even with a single processor. The 9 scheduling problems are solved to optimality and in this simple case, all the problems are feasible. In this case, debottlenecking is not necessary and the solution times for scheduling are 794 and 281 s for the serial and distributed runs, respectively. Problems DP2.1 and DP2.2 use the same problem description, with different discretizations employed to vary the size of the decomposed scheduling problems. DP2.1 employs a discretization of 15 to reduce the scheduling horizon to 100 h, while DP2.2 is solved using a discretization of 10, yielding a longer horizon of 150 h. The results show that the larger problems of DP2.2 results in an increase in solution time for the distributed case. DP2.2 could not be solved using a single processor due to the insufficient memory. The solution times for the DSP in the serial and distributed runs for DP2.1 and DP2.2 are ~1,000 and ~350 s, respectively. DP2.1 exhibits infeasibilities in the scheduling problems as does DP2.2. Therefore, the debottlenecking stage is executed, with one unit being added to make the problem feasible.

DP4 spans 18 aggregate time periods, instead of 9, as in the previous problems. The problem was solved with 1, 6 and 10 processors of equal speed (HP 9000/715 processors), and the results show that the single processor fails once again due to lack of memory space. The speedup achieved in the DSP solution is shown clearly by comparing the run times: 13,860 and 7,547 and 2,730 for the 1, 6 and 10 processor runs, respectively.

DP5 (Figure 8) exhibits behavior which leads to lower speedup of the distributed case due to the nature of the scheduling problems (one of the scheduling problems takes much longer to solve), as discussed earlier. The solution profile indicating the expansion of the plant capacity and the

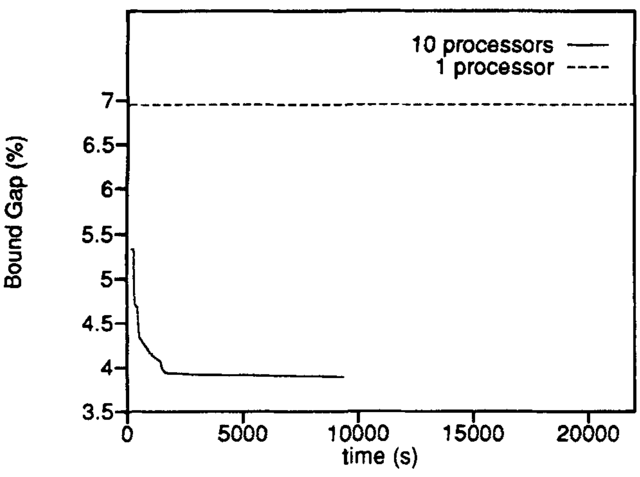

Figure 8. Progress of bound gap for problem DSP5.

corresponding production is shown in Figure 9. The single processor takes 3,065 s to solve the scheduling problems, while the distributed run is completed in 2,320 s. Noticing the idle time on the processors which finish early and wait for the limiting scheduling problem to terminate, the DTA mode described earlier was implemented. The result in Table 3 indicates that since the speedup is not equal to the number of processors employed, there is a scheduling problem which takes much longer to solve to optimality than the others. Implementing the DTA mode of the distributed run resulted in reducing the scheduling time to 1,317 s, as seen in Table 4. The DTA mode, therefore, enabled a considerable reduction in the computation time by utilizing the otherwise idle resources.

Conclusions

This article has addressed the development of a distributed algorithm for solving an important class of plant design and planning problems. Given the iterative nature of the design process, the ability to rapidly generate feasible solutions provides the engineer with the flexibility to explore a large number of designs. This enables the designer to choose between solutions that are economically equivalent but may differ radically from an implementation standpoint. In this context, distributed computing has proven successful in solving large instances of design problems derived from industrial case studies—some of which could not be solved using a single processor. The design algorithm described in this article uses multiple levels of parallelization and is a practical application of concurrent parallel branch-and-bound. With increasing emphasis on efficient plant designs and globalization of process operations, design and planning problems will continue to grow in size. Addressing these problems will require an increasing amount of computing power which is not likely to be delivered by faster processors alone. Distributed algorithm development frameworks such as DCABB are aimed to harness the power of computer networks to address these larger problems of practical interest.

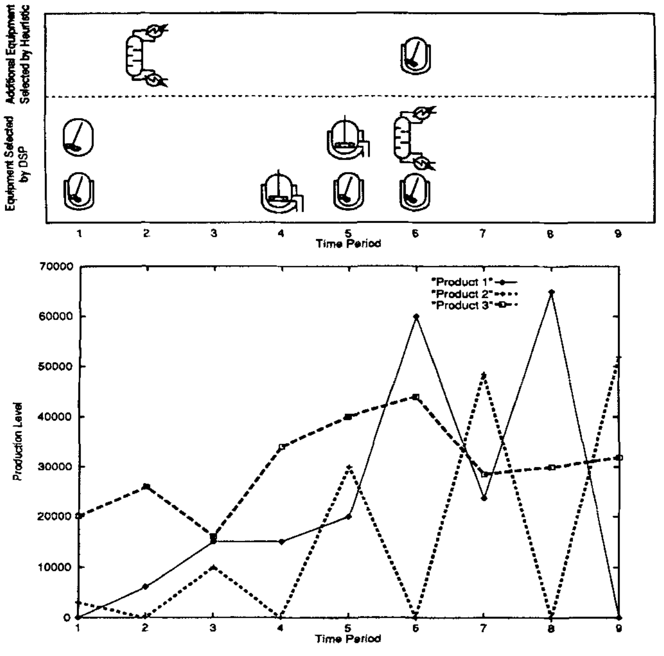

Figure 9. Solution profile of problem DP5.

Table 4. Distributed Computing Results Using STA and DTA Modes

Name	No. Procs.	DSP Time	No. SP	SP Time	DBP Time	Overall Time
	1	489	9	3,065	407	4,085
DP5	6(STA)	217	9	2,336	200	2,873
	6(DTA)	215	9	1,317	212	1,874

Notation

$A_{s\tau}^{\max}$	= maximum amount of resource s that can be stored at time τ
$A_{s\tau}$	= amount of resource s in inventory of time τ
$B_{ij\tau}$	= aggregate amount of material processed by task i on equipment j during time period τ
$C_{j\tau}^{\text{equip}}$	= cost of one unit of equipment j when purchased for subsequent use (in dollars adjusted for inflation and discounted to time period $\tau = 1$)
$C_{ij\tau}^{ov}$	= variable operating cost of task i on equipment j in time period τ
$C_{ij\tau}^{of}$	= fixed operating cost of task i on equipment j in time period τ
$C_{j\tau}^{mp}$	= manpower and other costs associated with operating unit j in time period τ
F_{si}^{cons}	= fraction of resource s consumed in one unit of output of task i
F_{si}^{prod}	= fraction of resource s produced in one unit of output of task i
H_{τ}	= length of time period τ
i	= task
I	= set of tasks
I_{j}^{tasks}	= index set containing indices for tasks which can be processed in equipment j
I_{sp}^{tasks}	= index set containing indices for tasks which produce resource s
I_{sc}^{tasks}	= index set containing indices for tasks which can consume resource s
I_{i}^{equip}	= index set containing indices for equipment types which can process task i
j	= equipment type
J	= set of equipment types
k	= scenario
K	= set of all possible scenarios

m_{ij} = capacity of equipment j during one batch when processing task i measured in units of total resource output upon completion of the batch (i.e., mass of output of task i on equipment j)

$n_{j\tau}$ = total number of units of equipment type j which will come into the production line at the beginning of time period τ

$N_{j\tau}$ = total number of units of equipment type j available for use at time τ

P_{ij} = processing time of task i in equipment type j

$P_{k\tau}$ = probability associated with a particular scenario k during time period τ

$q^b_{s\tau}$ = quantity of resource s bought at time τ

$q^{s0}_{sk\tau}$ = quantity of resource s produced and sold which is lower than the scenario demand level at time τ

$q^{s+}_{sk\tau}$ = quantity of resource s produced and sold which is above the scenario demand level at time τ

$Q^b_{s\tau}$ = maximum amount of resource s which can be bought at time τ from external sources

$Q^s_{sk\tau}$ = maximum amount of resource s which can be sold under scenario k at time τ for profit

s = resource/material

S = set of resources

T = set of all time periods

τ = time period

$v^s_{sk\tau}$ = value of resource s if scenario k is realized at time τ when sold to customers

$v^b_{s\tau}$ = value (cost) of resource s at time τ when bought from suppliers

$y_{ij\tau}$ = number of times task i is performed on equipment j at time τ

Literature Cited

Beasley, J. E., "Supercomputers and Operations Research," *J. of Operations Res. Soc.*, **38**(11), 1085 (1987).

Bitran, G. R., and A. C. Hax, "On the Design of Hierarchical Production Planning Systems," *Decision Sci.*, **28**, 28 (1978).

CPLEX, *Using the CPLEX Callable Library and Cplex Mixed Integer Library, version 2.1*, Cplex Optimization Inc., Incline Village, NV (1993).

Elkamel, A., "Scheduling of Process Operations Using Mathematical Programming Techniques: Towards a Prototype Decision Support System," PhD Thesis, Purdue Univ., W. Lafayette, IN (1993).

Elkamel, A., M. G. Zentner, J. F. Pekny, and G. V. Reklaitis, "Optimal Scheduling of the Resource Constrained Batch and Semi-Continuous Chemical Plant," Oper. Res. Soc. of Amer. meeting (1993).

Garey, M. R., and D. S. Johnson, *Computers and IntractabilityA Guide to the Theory of NP-Completeness*, W. H. Freeman, New York (1979).

Gendron, B., and T. G. Crainic, *Parallel Branch-and-Bound Algorithms: Survey and Synthesis*, No. 913, Centre de Recherche sur les Transports, Montreal (1993).

Grossmann, I. E., and R. W. H. Sargent, "Optimum Design of Chemical Plants with Uncertain Parameters," *AIChE J.*, **24**(6) (1978).

Ibaraki, T., "Enumerative Approaches to Combinatorial Optimization: I and II," *Annals of Operations Research*, **10** and **11** (1988).

Kindervater, G. A. P., and J. K. Lenstra, "Parallel Algorithms," *Combinatorial Optimization: Annotated Bibliographies*, M. O'hEigeartaigh, J. K. Lenstra, and A. H. G. R. Kan, eds., p. 106, Wiley, Chichester, U.K. (1985).

Kindervater, G. A. P., and J. K. Lenstra, "Parallel Computing in Combinatorial Optimization," *Annals of Oper. Res.*, **14**, 245 (1988).

Kocis, G. R. and I. E. Grossmann, "Computational Experience with DICOPT Solving MINLP Problems in Process Systems Engineering," Eng. Des. Res. Center Report EDRC-06-43-88, Carnegie Mellon Univ., Pittsburgh, PA (1988).

Kudva, G. K., "DCABB: A Framework for the Development of Distributed Branch and Bound Algorithms," PhD Dissertation, Purdue Univ., West Lafayette, IN (1994).

Kudva, G. K., and J. F. Pekny, "DCABB: A Distributed Control Architecture for Branch and Bound Calculations," *Comput. Chem. Eng.*, **19**, 847 (1995).

Mauderli, A., and D. Rippin, "Production Planning and Scheduling for Multi-Purpose Batch Chemical Plants," *Comp. Chem. Eng.*, **3**, 199 (1979).

Nemhauser, G. L., and L. A. Wosley, *Integer and Combinatorial Optimization*, Wiley, New York (1988).

Reinhart, H. J. U., and D. W. T. Rippin, "Design of Flexible Multiproduct Plants. A New Procedure for Optimal Equipment Sizing under Uncertainty," AIChE Meeting, New York (1987).

Reklaitis, G. V., "Progress and Issues in Computer Aided Batch Process Design," *Foundations of Computer Aided Process Design*, 1989 (1989).

Sahinidis, N. V., and I. E. Grossmann, "Multiperiod Investment Decision Model for Processing Networks with Dedicated and Flexible Plants," *Ind. Eng. Chem. Res.*, **30**(6), 1165 (1991).

Shah, N., and C. C. Pantelides, "Design of Multipurpose Batch Plants with Uncertain Production Requirements," *Ind. Eng. Chem. Res.*, **31**, 1325 (1992).

Subrahmanyam, S., M. H. Bassett, J. F. Pekny, and G. V. Reklaitis, "A Hierarchical Approach to Batch Plant Design," *Proc. Process Sys. Eng. Conf.*, Kyongju, Korea (1994a).

Subrahmanyam, S., J. F. Pekny, and G. V. Reklaitis, "Design of Batch Chemical Plants Under Market Uncertainty," *Ind. Eng. Chem. Res.*, **33**(11), 2688 (1994b).

Takamatsu, T., I. Hashimoto, and S. Shioya, "On Design Margin for Process Systems with Parameter Uncertainty," *J. Chem. Eng. Japan*, **6**, 453 (1973).

Vaselenak, J. A., I. E. Grossmann, and A. W. Westerberg, "An Embedding Formulation for the Optimal Scheduling and Design of Multipurpose Batch Plants," *Ind. Eng. Chem. Process Des. Dev.*, **26**, 139 (1987).

Wellons, H. S., and G. V. Reklaitis, "The Design of Multiproduct Batch Plants Under Uncertainty with Staged Expansion," *Comput. Chem. Eng.*, **13**, 115 (1989).

Wellons, M. C., and G. V. Reklaitis, "Scheduling of Multipurpose Batch Chemical Plants, 1. Formation of Single-Product Campaigns," *Ind. Eng. Chem. Res.*, **30**, 671 (1991).

Manuscript received Apr. 7, 1995, and revision received July 24, 1995.