DISTRIBUTED BRANCH AND BOUND ALGORITHMS FOR GLOBAL OPTIMIZATION

IOANNIS P. ANDROULAKIS* AND CHRISTODOULOS A. FLOUDAS†

Abstract. This paper presents computational results of the parallelized version of the αBB global optimization algorithm. Important algorithmic and implementational issues are discussed and their impact on the design of parallel branch and bound methods is analyzed. These issues include selection of the appropriate architecture, communication patterns, frequency of communication, and termination detection. The approach is demonstrated with a variety of computational studies aiming at revealing the various types of behavior of the distributed branch and bound global optimization algorithm can exhibit. These include ideal behavior, speedup, detrimental, and deceleration anomalies.

Key words. Global optimization, parallel computing, branch and bound.

1. Introduction. A wide range of optimal selection problems, in diverse scientific areas, can be formulated as *non linear constrained optimization problems*. One of the most common characteristics of these problems is the presence of *non–convexities* in their modeling representations. Non–convexities complicate solution methodologies since most existing optimization algorithms rely on identifying stationary points in the feasible space which satisfy the Karush-Kuhn-Tucker optimality conditions. These, except for problems of special structure, fail to discriminate between *global* and *local* solutions. Locating therefore the global minimum solution of a general non–convex optimization models is of primary importance. A common characteristic of all global optimization approaches is their increased computational requirement as the size of the problem increases. Towards this goal recently important advances have been reported and reviewed extensively, [13, 17, 15, 19]. In this work, we will focus on the αBB method which is based on a branch and bound framework coupled with a novel convex lower bounding function that is generated at each node, [5, 4, 3, 1].

Because the global optimization approach, αBB, is based on a branch and bound framework recent advances in the theory and practice of parallel algorithms for mathematical programming problems become more relevant, [23]. Furthermore, the concept of parallelizing branch and bound algorithms has created an enormous amount of scientific interest and has attracted various applications, [18]. The appearence of parallel computers in the world of scientific computing has already had a significant impact in the development of parallel global optimization algorithms, [24].

The scope of this paper is to present our recent findings in parallelizing the αBB algorithm and to present our computational experience in terms

*Corporate Research Science Laboratories, Exxon Research and Engineering Company, Annandale, NJ 08801,

†Chemical Engineering Department, Princeton University, Princeton, NJ 08544. Author to who all correspondance should be addressed.

of developing efficient distributed implementations of branch and bound algorithms. We will begin by presenting the key ideas and concepts behind the αBB so as to exemplify the need for increased computational efficiency via parallelizing the underlying branch and bound strategy. Subsequently, the key components of a distributed branch and bound algorithm will be discussed and our distributed implementation will be analyzed. One of the main contributions of this work is to identify the various computational behaviors of a distributed implementation of a branch and bound algorithm.

2. The branch and bound framework of αBB. The αBB has been developed so as to address the general twice differentiable non–convex optimization problem of the form:

$$\begin{aligned} &\min_x f(x) \\ &s.t. g(x) \leq 0 \\ &\quad h(x) = 0 \\ &\quad x \in X \subset \Re^n \end{aligned}$$

where $f, g, h \in C^2$, are twice–differentiable functions and x is a vector of size n.

The αBB operates within a branch and bound framework, whereby in each iteration the current domain is partitioned and a *convex lower bounding* problem is solved so as to generate *valid lower bounds* to the global minimum for each domain. Recent advances in the theory and implementation of the sequential αBB have made it one of the most attractive global optimization algorithms, [4, 3, 1]. The essence of the αBB algorithm is to derive for each non–convex term in the original formulation a convex underestimation of the form:

$$L(x) = f(x) + \sum_{i=1}^{n} \alpha_i (x_i^L - x_i)(x_i^U - x_i) \tag{2.1}$$

The determination of the α values is the most important step for the successful implementation of such a convex underestimating algorithm, [4]. The nature of the underestimation provided by the αBB sets it apart form other global optimization methods. The need for incorporating underestimators for terms of special structure, such bilinear or univariate concave, was also identified early on and is incorporated, [5]. The αBB algorithm proceeds in a branch and bound framework shown in Figure 1. Recent developements, [2], have expanded the scope of αBB so as to address problems involving continuous as well as discrete variables.

A branch and bound algorithm is also referred to in the computer science literature as a *divide and conquer* approach and this describes quite accurately the idea behind branch and bound. The main characteristic of this

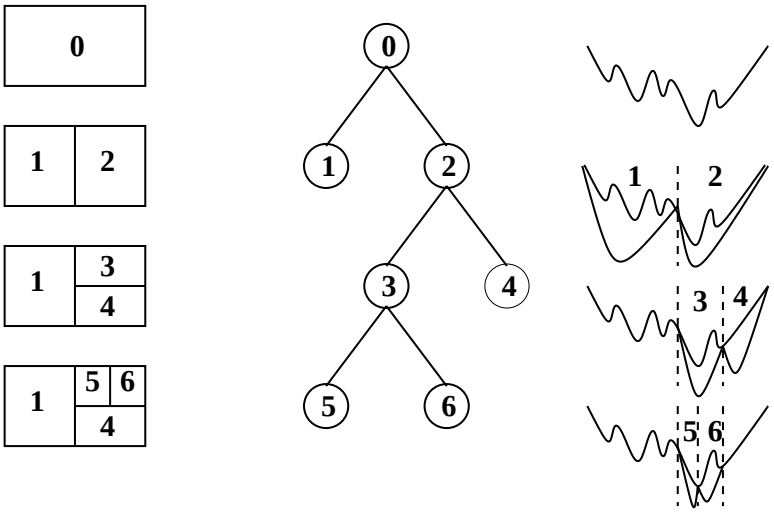

FIG. 1. *Branch and Bound framework.*

class of algorithms is that *as the size of the domain is reduced the quality of the representation improves.* Clearly, from equation (2.1) we can observe that the smaller the domain, that is, $x_i^U - x_i^L$ becomes smaller, the tighter the quadratic underestimation becomes. This is one of the properties of the quadratic underestimation $L(x)$ that is being proposed and is described in greater detail in [22]. Clearly, one is tempted to propose that: if a finer approximation be used from the early stages then a better domain characterization should be produced. This is indeed the case, and this is an idea which has already been incorporated within the αBB framework, [6]. Because finer initial domains result in better approximations, exploring simultaneously multiple domains should result in a more efficient search. This is the rationale behind advocating the development of a parallel search, in other words a distributed implementation of the branch and bound-based αBB algorithm. In this case, the characterization of each domain would correspond to the solution of the convexified non–convex problem within the domain of interest.

The key characteristics of any branch and bound algorithm are the various branching and bounding phases. We will assume for the purpose of this paper that these have been decided. At each node of the αBB the mathematical structure of the underestimating problem solved remains the same. The only difference between problems is the domain within which they are being solved. These define the underestimation of each domain. Decisions related to branching variables and lower bound estimation will not be discussed here. For our purposes, branching is performed along the variable that participates in the non–convex term with the largest un-

derestimation. A thorough analysis of branching schemes within the αBB algorithm is presented in [1]. The objective in the following sections is to discuss the critical issues that pertain exclusively to the distributed implementation of the branch and bound algorithm.

3. Critical Issues. This section discussses the various critical aspects which become important when one wishes to implement a branch and bound framework in a distributed computing environment. Our scope is not to discuss those issues which are important in a sequential implementation of a branch and bound algorithm, but rather to focus on implementational issues which become critical when branch and bound is embedded in a parallel computing environment. As will become apparent, it is very difficult to identify *the* optimal distributed implementation of a branch and bound algorithm.

The key issues are discussed in the following:

- **State Space Representation**: This is a fundamental decision one has to make prior to designing the distributed implementation. One has essentially to decide between a *static* or a *dynamic* representation of the states of the problem, [20]. In the former, an initial partitioning is postulated, the corresponding nodes of the branch and bound tree are, somehow, partitioned among the various processors and each processor works exclusively with its assigned pool. In Figure 2 we depict a hypothetical situation in which the initial search domain is partitioned into 4 major sub–domains (these can either be a single domain or collection of simpler ones) and each processing node executes the successive steps of a branch and bound algorithm on its assigned domain(s). Each node detects locally its termination based on some criterion. The number of nodes visited and corresponding CPU time per processing node is a function of the initial domain(s) assigned. As a results, processing node 1 will require a total of t_1 time units before it declares convergence and terminates. Node 2 will require t_2 units and so on. The overall algorithm is rate–limited by the rate of the slowest processing node, which in that case happes to be 4. Therefore, regardless of how fast the other nodes terminated, each of them remains for a period of $t_4 - t_j$ time units in an idle state.
  As can be seen from Figure 2, this type of partitioning runs the risk of forcing the processing elements to the so–called state of *starvation* in which processing nodes remain idle. This is the result of the uneven distribution of work–load which is the result of the nature of the problems initially assigned to each processing node. The termination of each processing node simply depends on the initial problem assigned. In general, similar schemes are highly inefficient and therefore rarely used. The communication is non–existent (the results are collected at the end and the best solution

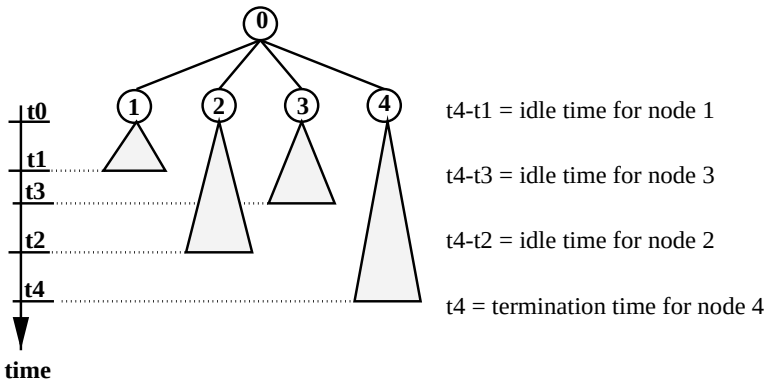

FIG. 2. *Static Distribution of Sub–problems.*

is reported). Under certain conditions this is the most suitable problem decomposition and only then it is advisable, [8]. In most cases a *dynamic* representation of the state space is adopted. In other words, instances of the problem are created dynamically and are being assigned to processing nodes so as to minimize the idle time of each processing element, provided that care has been taken to guarantee an equally distributed work–load. In such a case, one has to make a choice on how to store the created problem that is to be solved. One can assume a *centralized* scheme in which a processing element (single processor or collection of processors) is responsible for maintaining the queue of problems to be solved. A very appealing implementation of such a scheme is described in [25]. The so–called *processor shop* concept is introduced whereby processors are viewed as independent actors free to choose their activities which include not only which problems to solve but also the type of branching rules that are to be used, and the type of the underestimation as few instances. A simple picture of how the centralized pool operates is presented in Figure 3.

One can draw a comparison to a *blackboard* approach whereby problems awaiting to be solved reside in a memory location accessible to all processing nodes. In [25], a careful design of the algorithm guarantees the integrity of the queue of problems by restricting access to one processor at a time. Such an approach is advisable only for problems in which the communication overhead associated with processors waiting to access the queue of problems is insignificant compared to the computation time.

In a *decentralized* representation scheme, the pool of problems waiting to be solved is stored locally in each processing node as new instances are being created. In such a case, an initial partitioning

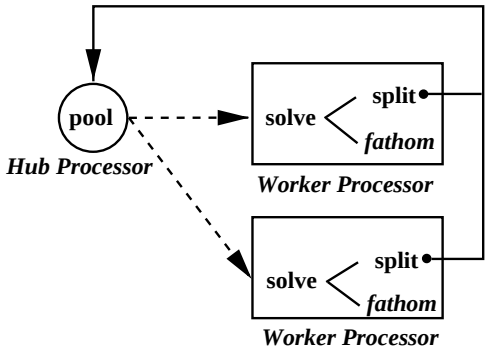

FIG. 3. *Centralized Representation of Problem Instances.*

is performed and subsequently each node has access to a sub–space of the state space. The fundamental difference between the distributed and the static representation is that in the latter a constant flow of information exchange, in the form of upper/lower bounds and problem instances, guarantees that the extreme conditions of starvation will be avoided. Figure 4 depicts such a scheme. Various important implementational details have to be taken into account and these will be discussed in the sequel since this is the scheme that was adopted in our approach.

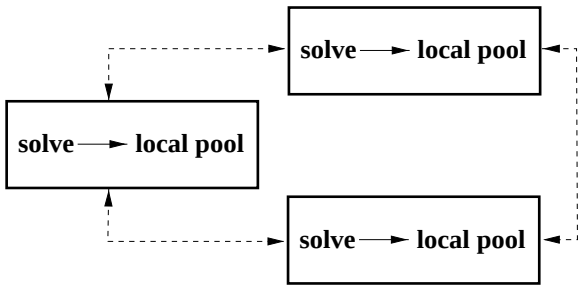

FIG. 4. *Decentralized Representation of Problem Instances.*

- **Nature of the Active Pool**: Up to this point, we have discussed the concept of *storing* problem instances. A critical design decision concerns the *state* of the stored problems. These can be stored as either *solved* or *unsolved*, [12]. Both have their advantages and some researchers have actually incorporated both queues in their developments, [25]. A solved problem is one which has been solved for its lower bound and that value, if it survives fathoming, is assigned to be the corresponding lower bound. An unsolved problem is one that inherits the parent node's lower bound. It is actually

hard to predict which one is the most efficient. The unsolved queue for instance can be very efficient if at some point the parent node is to be fathomed. So will the children nodes without having to actually be solved for their corresponding lower bounds. On the other hand, savings in memory can be achieved with a solved queue since immediately one detects the quality of the lower bound of a node and this node is stored if and only if it survives fathoming.

- **Degree of Synchronism**: Once the character of the stored problems has been decided and the way they will be stored is set, one has to decide on how they will be accessed, that is the degree of *global control* that will be enforced. Various combinations of the character of the pool and the degree of synchronism give rise to different models, [18]. Global synchronism guarantees ordered access to any memory location but might result in serious communication overhead. From an implementation point of view, lack of global control results in easier implementations and faster information exchange but can result in wasteful computation and in some instances, depending on the type of problem solved, it can result in an unpredictable and spurious behavior, [7]. Spurious behavior in terms of the final result is excluded within the branch and bound framework. The worst case scenario is that more nodes than needed will be explored. Nevertheless, computational *anomalies*, [21], can be developed and it is our aim to identify and present those with our computational results.

  It is widely accepted that the computational requirement for any task is proportional to the number of problem states in the search tree. Unlike other mathematical operations, such as matrix inversion, a concurrent search within a branch and bound framework does not reduce the computational requirements in linear or even predictable way. Unlike discrete domains, continuous ones are not characterized by a finite state representation. One can derive such a representation but a certain tolerance has to be defined. The result of that is that distributing the work load can have unpredictable results. As can be seen if Figure 5 one can encounter either *detrimental anomalies* or *speedup anomalies*. In the former case, the partitioning, distribution and subsequent diffusion of the state space instances generates additional domains which are being explored. In Figure 5, N_{seq} is defined as the number of branch and bound nodes that will be required by the sequential execution of the algorithm, whereas N_{par} is the number of branch and bound nodes required by the parallel implementation. The simultaneous characterization of these nodes results in actually solving more branch and bound nodes that would have been solved sequentially thus resulting in a *detrimental anomaly*. On the other hand, one can envision a situation in which a node that was not to be se-

lected, because of the quality of the lower bound, for expansion in a sequential framework, is indeed selected in the distributed framework and is actually producing a very tight upper/lower bound that helps is reducing the size of the tree. In such a case the number of expanded nodes of the parallel algorithm is smaller than the corresponding number of the sequential algorithm thus leading to *speedup anomalies.*

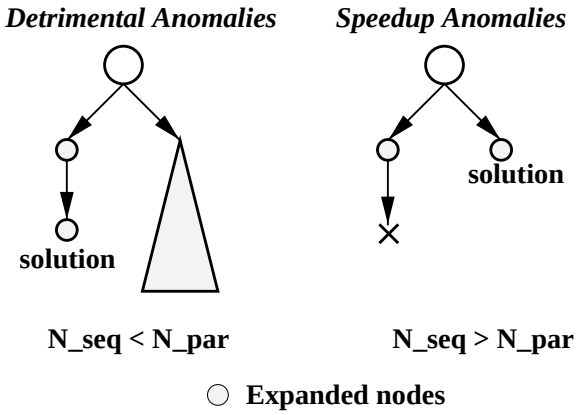

FIG. 5. *Computational Anomalies.*

- **Communication Patterns**: In any type of computation that is based on message passing communication patterns, *origin, destination* and *frequency of communication exchange* are very crucial. These patterns aim at diffusing computational work within the computer network. The importance of such a design aspect is to avoid the localization of the search of the branch and bound tree in terms of a single processor doing most of the useful work, as well as proceeding in a depth first fashion as opposed to a breadth first, [26]. The structure that is embedded within the topology of the computer network is also crucial in allowing faster diffusion through local information exchange, [27]. This last point is very important since our target is to develop a distributed framework that is exclusively based on local information exchange.
- **Termination Detection**: Detecting termination of a distributed algorithm that possesses no global control is a highly non–trivial task, [9]. In any centralized algorithm, either sequential or distributed with global control, a local termination criterion can be derived and based on that one can detect convergence. In an asynchronous environment two criteria have to be satisfied: a *local* one based on which each processing node has detected local convergence, and a *global* one based on which no communication messages are pending in the network in terms of unsolved problems in

transit from one processing element to another or in terms of upper/lower bound values. Intricate specialized algorithms had to be devised that would detect whether messages are pending following local termination detection, [10, 11].

All the aforementioned issues are very crucial design decisions and in Section 4 they will be discussed in more detail so as to justify the specific selections that were made and their impact on the computational performance. The discussion will be accompanied by computational results so as to support the selections and also to emphasize the effect of them.

4. Designing a Distributed Implementation.

4.1. Architecture. The first issue we address is the architecture to be used. By architecture we do not refer to the architecture of the parallel machine, but rather the virtual one which will define the communication protocol of our algorithm. There are two key characteristics we would like to achieve: (i) asynchronous computation/communication with a distributed queue of solved problems, and (ii) non–symmetric diffusion of high priority work. Schematically, Figure 6 shows the key characteristics of the proposed architecture. We embeded a *ring* architecture onto the mesh of the **Intel-i860** Paragon of the U.C. San Diego Super-computing Center. Each node maintains its own local list of solved problems and it periodically communicates to other nodes information such as upper/lower bounds and problem instances.

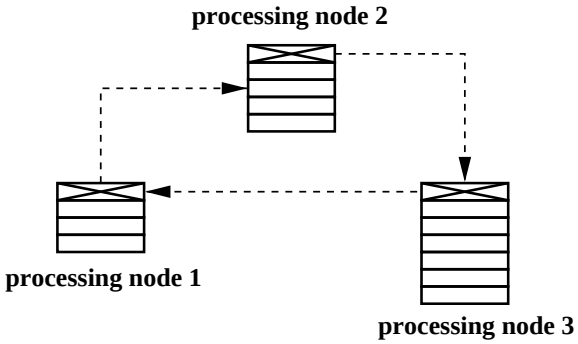

FIG. 6. *Model ring architecture.*

One key issue to be addressed is the communication pattern that is used for message passing. In other words, the origin and destination of each message. Nearest neighbor communication defines a simple *ring*. Messages move from node (i) to node $(i+1)$. Such local communication patterns can result in a very localized way of computation. This can easily result in either a depth first search tree or uneven work load distributions. In order to break such symmetries an *asterisk* can be defined instead of a simple ring.

Based on the discussion of Section 3, we designed a distributed architecture that exhibits distributed representation of the state space in the form a local queues of solved problems while the computation and communication exchange are asynchronous.

4.2. Patterns and Frequency of Communication. Communication Patterns: The concept of a neighbor need to de redefined. Each node (i) communicates messages to node $(i + 2log_2(P))$ for a network of P processors. The result for a system of $P = 8$ nodes is depicted in Figure 7. The objective is to break any patterns of symmetry and avoid strictly nearest neighbor communication. Furthermore, we define a *time–varying* asterisk in which the destination of a message is shifted each time a new message originates from node (i), as shown in Figure 8.

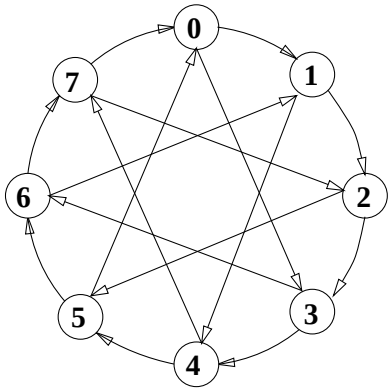

FIG. 7. *Origin/Destination of time varying Asterisk at time t.*

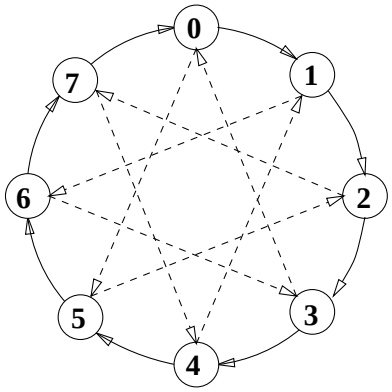

FIG. 8. *Origin/Destination of time varying Asterisk at time* $t + 1$.

Based on the example of Figures 7 and 8, the initial structure of the

asterisk, shown in Figure 7, is $0 \to 3 \to 6 \to 1 \to 4 \to 7 \to 2 \to 5 \to 0$. In other words, node 0 transmits a message to node 3, node 3 transmits to node 6 and so on. The next instance the structure, shown in Figure 8, becomes: $0 \to 5 \to 2 \to 7 \to 4 \to 1 \to 6 \to 3 \to 0$. It should be pointed out that in our implementation the computation/communication steps for each processor are performed asynchronously and therefore the destinations change independently.

In order to illustrate the performance of a ring architecture with nearest neighbor communication only ($i \to i+1$) and the time–varying asterisk, Figure 9 depicts a typical work–load distribution for each of the architectures on a 32–node environment.

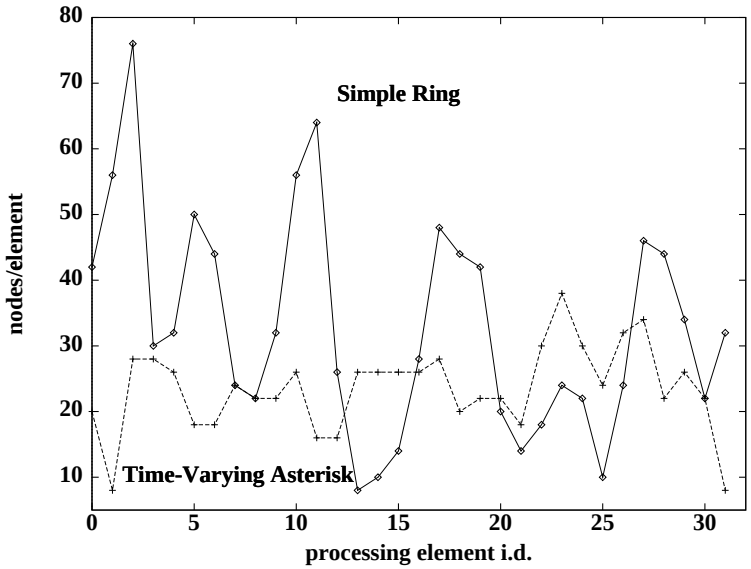

FIG. 9. *Comparison of simple ring and time–varying asterisk.*

We observe that the time–varying asterisk achieves a better work–load distribution among the processors as seen from Figure 9. The simple ring assigns an average of 33 problems per node with a standard deviation of 16, whereas the time–varying asterisk assigns 23 with a standard deviation of 6. This is a first indication of how an architectural decision can actually affect the overall performance of the execution of the algorithm.

Communication Frequency: Apart from the origin and destination of the exchanged messages it is also important to study the effects of the frequency of information exchange. At this point we will elaborate on the type of information that is being exchanged. We will postulate that each node communicates to its neighbors (based on our definition) with the *highest priority work*, which is the node in a local solved queue that has the best lower bound. This heuristic aims at minimizing the probability

of a single node performing most of the useful work by operating in a best first mode locally. Rare information exchanges are known to cause performance deterioration, [26], since if information exchange does not take place frequently, then the useful parts of the branch and bound tree are located in a single processor, and all useful work is limited. As can be seen from our computational results, shown in Figure 10, the work load distribution becomes more uneven as the interval, measured in terms of iterations, increases. The maximum peaks of the curves imply that certain nodes perform more work than others thus certain nodes perform local best first searches while others exhaust their list and might even remain idle for long periods of time.

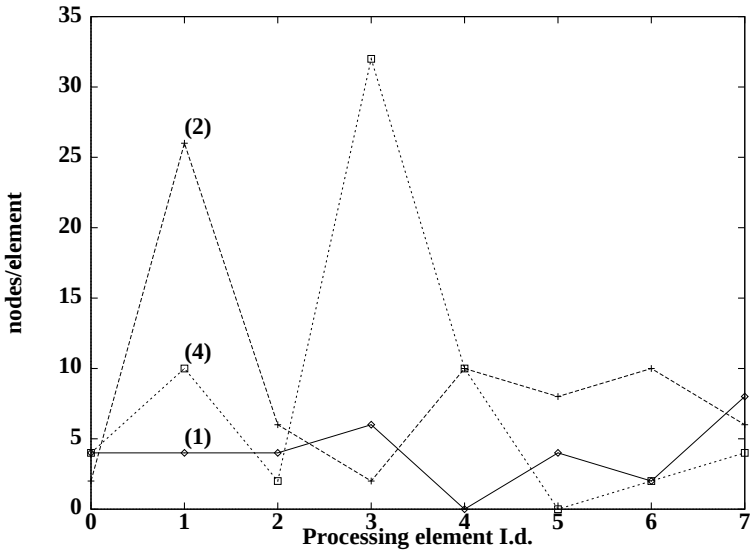

FIG. 10. *Effect of communication frequency on work–load distribution.*

4.3. Queue Initialization. In principle there are two ways one can initialize a branch and bound approach. One can either begin from the root node and successively branch and create new instances by solving the resulting problems. This is clearly a very inefficient way of operation since for the induction period, until the number of solved nodes equals the number of processing elements, there exist idle processing nodes. This is the reason why a number of partitions is created at the start of the algorithm and these problems are distributed among the processors. We choose to distribute the problem instances randomly so as to avoid the additional bias of single nodes having in its initial pool of unsolved problems the area surrounding the solution. Furthermore, a decision has to be made regarding the number of branch and bound nodes that will be created at the beginning. It will be shown that even this minor issue can have

an impact on the performance. As seen in Figure 11, for every problem instance one can identify similar distributions.

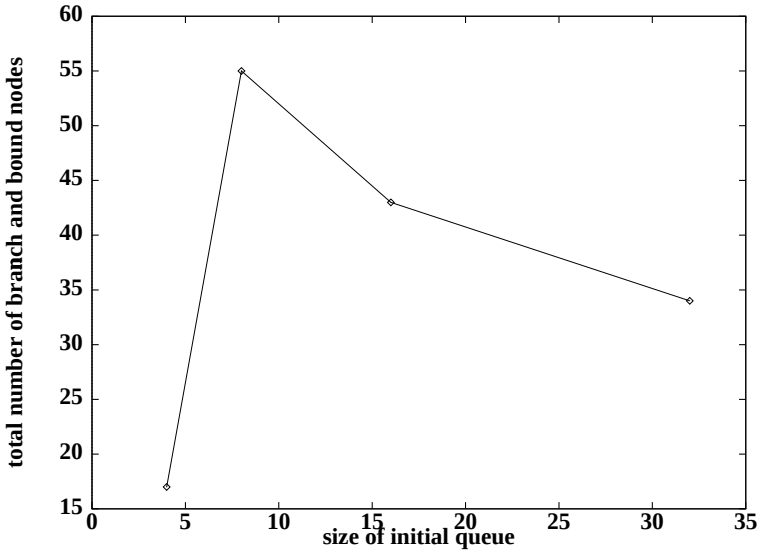

FIG. 11. *Effect of initial queue size.*

We consistently observed that small size initial queues result in better performance. The reasoning behind such a selection is that an effort has to be made to minimize the amount of wasteful work. It is not necessarily true that if one begins the search with a very fine partitioning the best computational results will also be recovered.

4.4. Termination. Determining the termination of an asynchronous algorithm is a non–trivial task. There are two requirements that have to be satisfied. Each node has to detect *local* termination (either *local upper bound - local lower bound* $\leq \epsilon$, or *local lower bound* $>$ *local upper bound*, or the local queue is empty (note that the first two conditions can be true since a node's local lower upper bound need not be generated locally by the node in question), and the system has to detect *global* termination (no messages are pending within the system). If the following two conditions are true:

1. bounded communication delays, $B < \infty$, and
2. no processors die

then *Dijkstra's* algorithm, [10], guarantees that correct termination of the distributed computation can be detected. The algorithm assumes that the processing elements can communicate a special *token* to their adjacent neighbor. The steps of the algorithm are as follows:

Step 0: Processors start working and P_0 has the *token*

Step 1: When P_0 becomes *idle*, P_0 sends *white* token to P_1
Step 2: If P_i sends work to P_j, $i > j$, then P_i becomes *black*
Step 3: If P_i has the *token*, and P_i is *idle*, (i.e., local termination),
then P_i passes token to P_{i+1}
If P_i is *black* then the *token* sent to P_{i+1} is *black*
If P_i is *white* the *token* is passed *unchanged*
Step 4: If P_i passes the *token* P_i becomes *white*

- **when P_0 receives white token the algorithm terminates**

As shown in Figure 12, two types of messages are communicated throughout the network of processing elements. Messages related to the problem, such as upper and lower bounds, which follow the time-varying asterisk pattern, and also messages carrying the *token* which are passed to the nearest neighbor. Each node, is also testing the steps of the algorithm that was just outlined and once global convergence is being detected a global termination message is being issued.

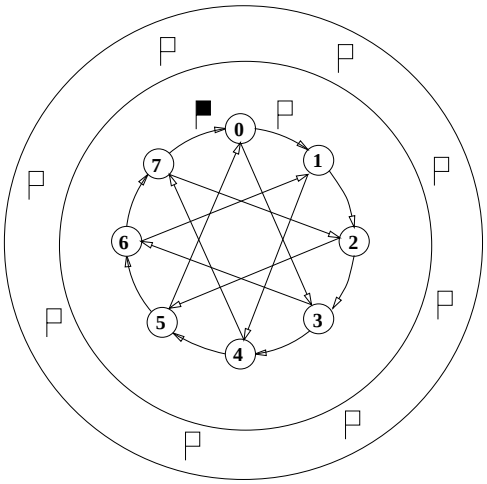

FIG. 12. Dijkstra's *termination algorithm.*

It should be noted that in our implementation there exist two independent paths of communicated messages. One route is reserved for the token to detect termination, and the other route is defined by the communication patterns of the time-varying asterisk.

4.5. Distributed Implementation. Having discussed the important points of a distributed implementation, we will now summarize the key features of the distributed branch and bound algorithm that is proposed:

- The algorithm was developed on the **Intel–i860** parallel machine of the University of California, San Diego. It is a mesh architecture

of 400 nodes.

- The state space representation was on a distributed queue of solved problems. Each node maintains a local queue.
- The queues are initialized simultaneously but care is taken so as to assign to processors initial non–contiguous domains.
- Processors diffuse high priority work (data structures containing the description of the domain generating the best lower bound) at each iteration based on patterns defined by the time–varying asterisk.
- Processors purge their local list as soon as the upper bound is updated.
- Termination is detected by the *Dijkstra's* algorithm.
- The computation/communication steps are totally asynchronous.

In Section 5, computational results are presented that exemplify the interesting phenomena occurring in an asynchronous implementation of a branch and bound algorithm.

5. Computational Results.

1. **Ideal Behavior**: The problem we address here deals with the optimization of reactor-separator-recycle network. We postulate a superstructure [1], as shown in Figure 13, and the objective is to identify the optimal configuration which corresponds to the global minimum of the cost function. This problem is a variation of problem 10.2 in [16] and the general formulation is described in greater detail in [14]. It describes a process of obtaining *mono* and *dichloro benzene* from *benzene* by allowing for the recycle of unreacted *benzene*. The mathematical formulation consists of 40 variables and 58 constraints and is presented in the Appendix.
   Figure 14 shows the number of branch and bound nodes and the total CPU required. As it can be seen, the total number of nodes remains practically constant, thus resulting in a substantial decrease of the total CPU time. This is precisely the ideal behavior. In this case, distributing the work-load results in a reduction of the CPU, because the total work-load remained the same. The problem was solved using 1, 4, 8, 16, and 32 processors. The optimal configuration would probably correspond to 16 processors since passed that point the additional overhead diminishes the improvement in terms of overall CPU time. The achieved speedup, S_p, and efficiency is shown in Table 4. The optimal structure is depicted in Figure 15.
2. **Speedup Anomaly**: This example is problem 2.7 in [16] and belongs to the class of *concave quadratic optimization problems*. It is

[1] A *superstructure* is defined as the set of all possible process alternative structures of interest which are candidates for feasible of optimal flowsheets. The superstructure features a number of different process units and their interconnections [14]

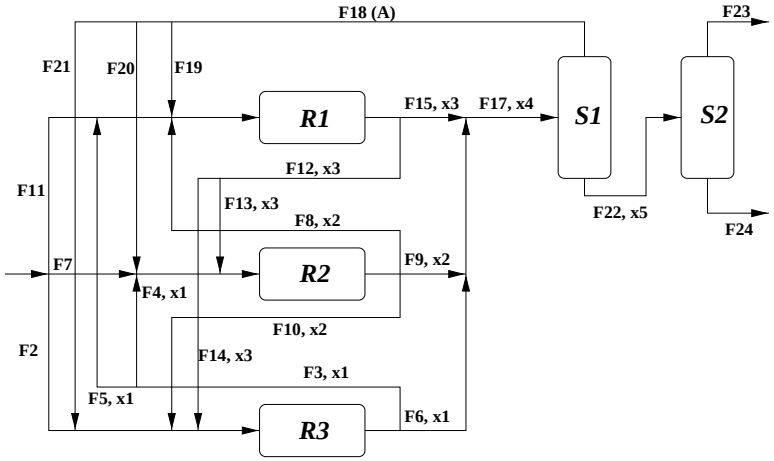

FIG. 13. *Reactor/Recycle/Separator System Superstructure. F's represent total flow rates and x's mass fractions.*

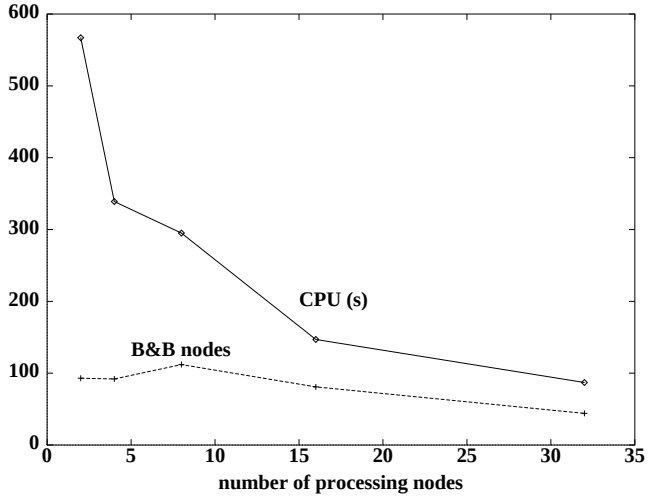

FIG. 14. *Computational results indicating* ideal behavior.

TABLE 1
Speedup and Efficiency of ideal behavior.

P	T	S_p	N	I_p
1	93	1.00	567	1.00
4	92	1.67	339	1.02
8	112	1.92	295	0.83
16	81	3.86	147	1.15
32	44	6.52	87	2.11

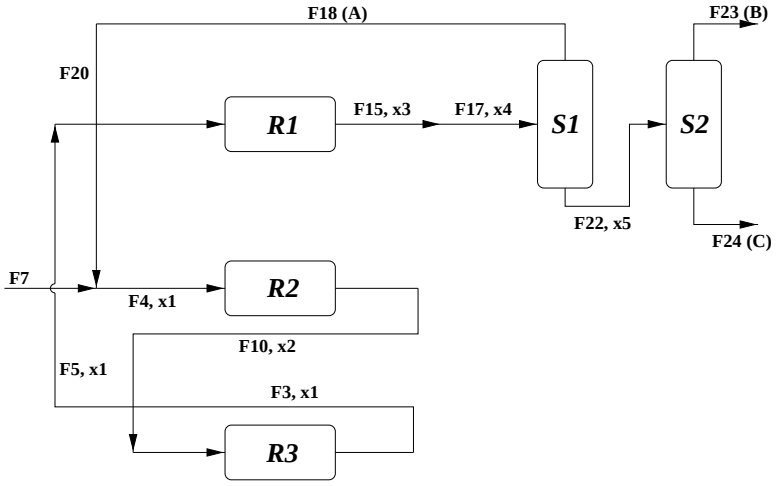

FIG. 15. *Reactor/Recycle/Separator System Optimal Structure.*

described by the following formulation (more details are presented in the Appendix):

$$\min \quad -0.5 \sum_{i=1}^{20} \lambda_i (x_i - \alpha_i)^2$$
$$s.t.\ x \in R = \{x : Ax \leq b, x \geq 0\} \subset \Re^{20}, \quad b \in \Re^{20}$$

The distributed version of our branch and bound actually visits fewer nodes than a sequential execution of a branch and bound algorithm. As seen in Figure 16 the results for 1, 2, 4, and 8 nodes suggest that from 1 to 4 nodes the number of branch and bound nodes visited drops. The 8–node configuration visited a few more nodes that have no significant impact.
The CPU time is not affected in an equally dramatic fashion due to the overhead penalties. We would like to point out that we are more concerned with the number of branch and bound nodes visited as a true measure of the response of the system rather than the machine and implementation dependent CPU time. Therefore, the anomalous behavior is reflected on the number of branch and bound nodes that drastically drop as we move away from the sequential algorithm.

3. **Detrimental Anomaly**: This next example is a variation of problem 9.2 in [16] and is also described in a generic form in [14]. It involves a system of reactors where the *Van der Vusse* reaction takes place (see Appendix). A superstructure is postulated, Figure 17, and the configuration that maximizes the production of

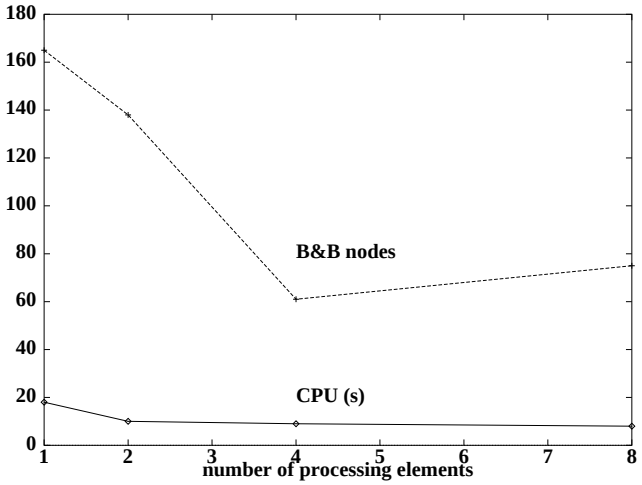

FIG. 16. *Computational results indicating* speedup anomaly.

TABLE 2
Speedup and Efficiency of speedup anomaly.

P	T	S_p	N	I_p
1	165	1.00	19	1.00
2	138	1.96	10	1.90
4	60	2.75	9	2.11
8	78	2.12	8	2.38

the useful product is identified. This examples represents the first type of unusual behavior. In this case the number of nodes visited by the parallel branch and bound actually increases with the number of visiting nodes. The mathematical formulation consists of 29 variables and 45 constraints and is presented in detail in the Appendix.

As shown in Figure 18, the number of problems solved increases but the additional nodes used reduce the overall CPU time. The reason for the seemingly improved behavior in terms of the CPU requirement, is understood by examining the work–load distribution in Figure 19.

We were able to maintain a well–balanced network in terms of the total number of problems that were eventually solved by each processing element. The fairly equal distribution of the work–load resulted in the reduction of the CPU time.

Clearly one observes that although the speedup improvements are substantial the efficiency of the algorithm is reduced as more and more problems are actually required. It so happens that the path

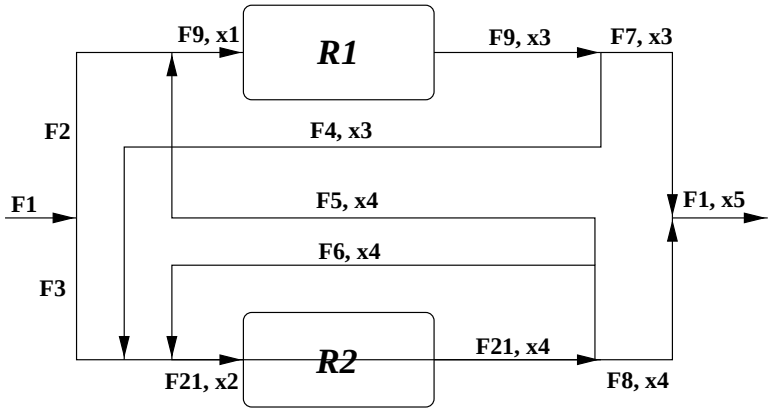

FIG. 17. *Reactor Network Design Problem Superstructure.*

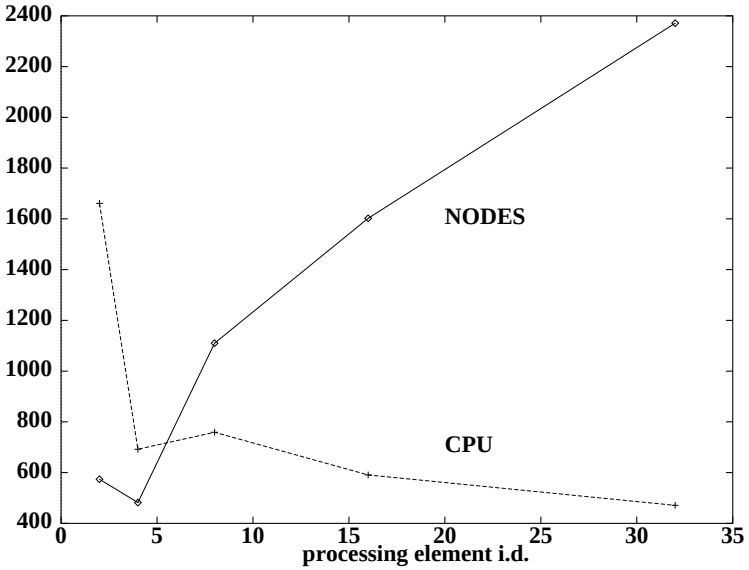

FIG. 18. *Computational results indicating* detrimental anomaly.

TABLE 3
Speedup and Efficiency of detrimental anomaly.

P	T	S_p	N	I_p
1	1630	1.00	590	1.00
4	690	2.36	480	1.23
8	720	2.26	1110	0.53
16	605	2.69	1605	0.37
32	460	3.54	2360	0.25

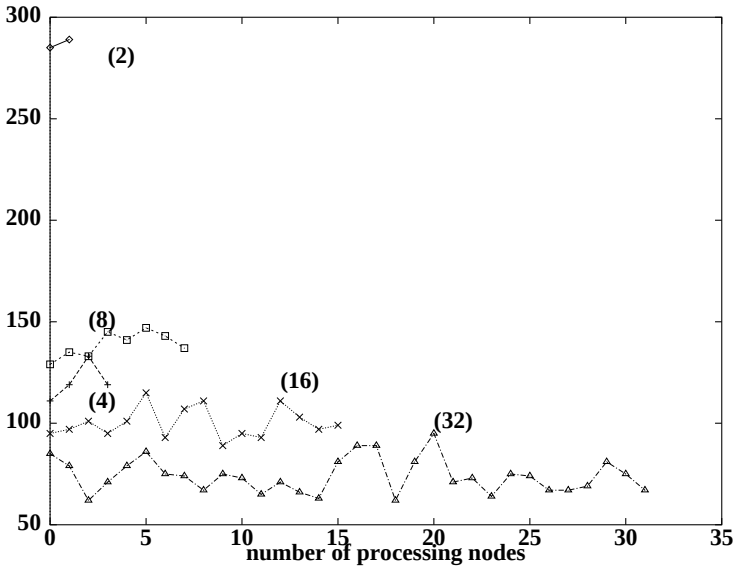

FIG. 19. *Equal work–load distribution.*

followed by the distributed branch and bound is such that the equal work–load distribution alleviates the burden of the increased number of state spaces that had to be visited. The optimal structure is shown in Figure 20.

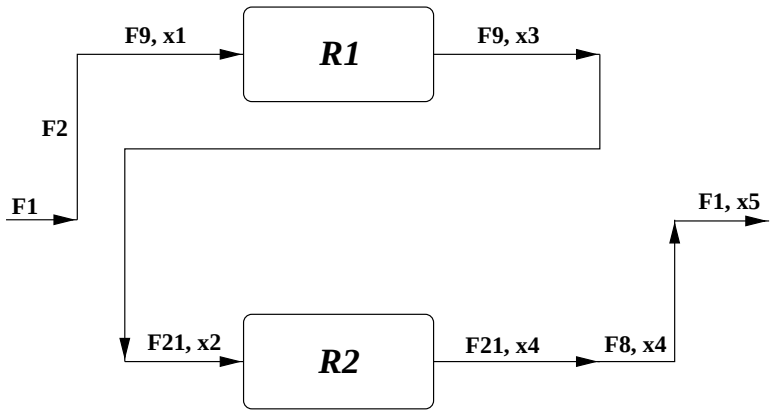

FIG. 20. *Reactor Network Design Problem Optimal Structure.*

4. **Deceleration Anomaly**: This last example is similar to the first one, where only the original superstructure has been augmented by an additional reactor in parallel. The system is described in Figure 21. The mathematical formulation is described by 41 variables

and 67 constraints. A detailed presentation of the mathematical formulation is presented in the Appendix.

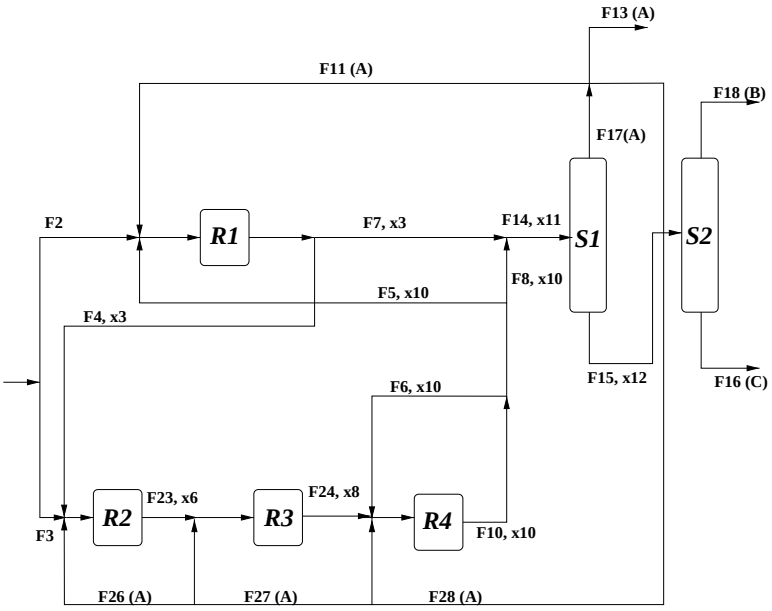

FIG. 21. *Reactor/Separator/Recycle System Superstructure.*

This example exhibits the so–called *deceleration anomaly* which is the situation in which the parallel branch and bound is actually taking more time than the sequential version as can be seen in Figure 22. This is the type of behavior one wishes to avoid. In this case the actual CPU time increases while the total number of nodes actually decreases. The only way this behavior can be explained is if severe unbalancing occurs in the way the problems are being distributed among the processors. indeed this is true should one observe the work–load distribution in Figure 23. Unlike the previous case, the uneven distribution of domains per processing node does not allow for computational improvements in terms of algorithm speedup. The optimal structure is shown in Figure 24.

6. Conclusions. This paper dealt with the distributed aspects and computational performance of a novel global optimization algorithm, αBB, which is based on a branch and bound framework coupled with a convex lower bounding functions. The critical issues related to distributed implementations of branch and bound algorithms were identified. These were: the state space representation, the nature of the active pool, the degree of synchronism, the communication patterns, and the termination detection. Specific design guidelines were presented so as to achieve the maximum per-

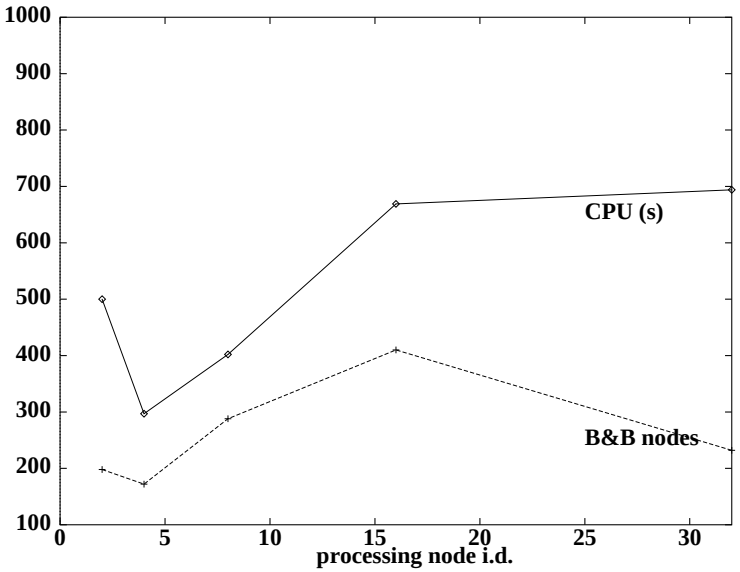

FIG. 22. *Computational results indicating* deceleration anomaly.

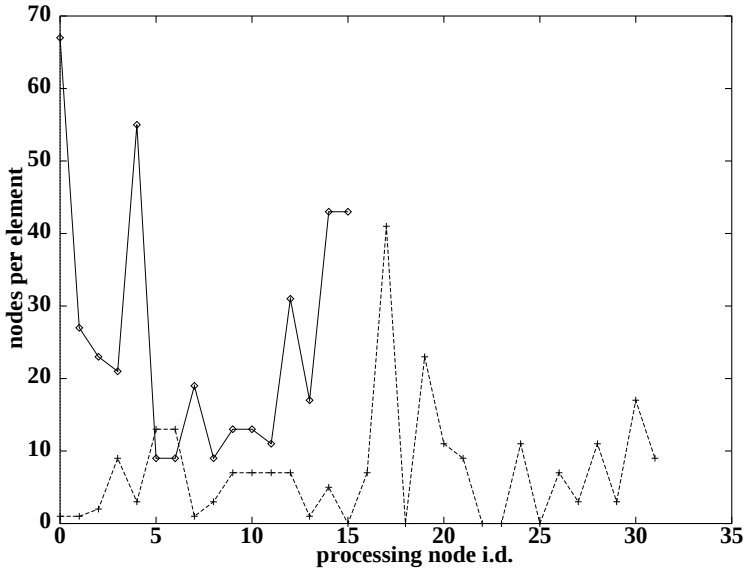

FIG. 23. *Unequal work–load distribution resulting in deceleration anomaly.*

formance of the implementation. Regarding the computational behavior, four global optimization problems were examined and the wide spectrum of computational behaviors were presented. These include: ideal behavior, speedup anomalies, detrimental anomalies, and deceleration anomalies. The response of the distributed implementation of a branch and bound al-

TABLE 4
Speedup and Efficiency of deceleration anomaly.

P	T	S_p	N	I_p
1	502	1.00	200	1.00
4	298	1.68	180	1.11
8	400	1.26	290	0.69
16	680	0.74	390	0.51
32	690	0.73	240	0.83

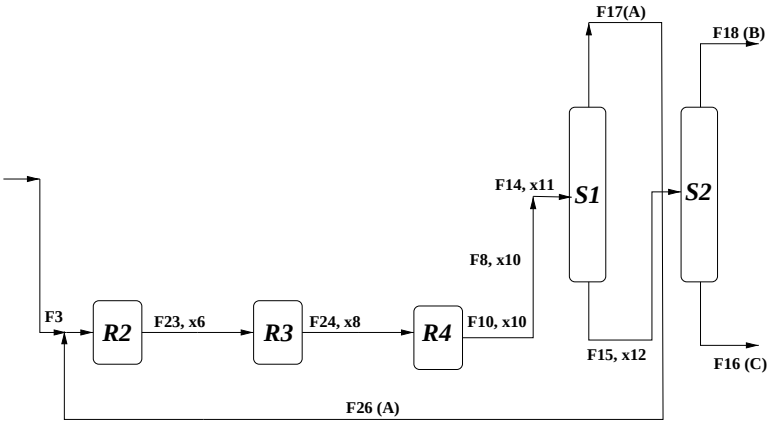

FIG. 24. *Reactor/Separator/Recycle System Optimal Structure.*

gorithm was found to be dependent on the inherent characteristics of the problem that is solved.

Acknowledgment: The authors gratefully acknowledge the support from the Air Force Office of Scientific Research, the National Science Foundation,the National Institutes of Health, and the National Supercomputing Center at the University of California, San Diego.

APPENDIX

For completeness purposes, we present the detailed formulations of the examples solved. The first, third and fourth problems have quite complicated mathematical formulations. In order to present those, we explore an additional capability of the computational implementation of the αBB algorithm which allows for the automatic generation of the complete mathematical model at run–time, [1]. The computer implementation generates a detailed model, with the original variable names which, as will be seen, is extremely useful to the user. More specifically the problems addressed

were the following:

- Example 1 of Section 5.
  This problem, corresponds to a variation of problem 10.2 in [16]. It describes a *benzene chlorination process*. It can utilize up to three reactors, of unknown size, and up to two separators. The superstructure allows for various possible interconnections between the reactors and the separators. We optimize an economic objective describing:
  - a reaction process:

$$C_6H_6 + Cl_2 \rightarrow C_6H_5Cl \ + \ HCl$$
$$C_6H_5Cl + Cl_2 \rightarrow C_6H_4Cl_2 + HCl$$

  - a separation process: the components to be separated consist of *benzene* (A), *mono-chloro benzene* (B), *di-chloro benzene* (C)
  - recycling components: *Benzene* (A).

  The detailed mathematical formulation, as it is generated automatically by αBB, is as follows:

```
Minimize
  obj = (((((((147.62 * f14) * x4a)
+ ((-445.544 * f17) * x4b))
+ (( 2793.79 * f22) * x5b)) + ((190.114 * f17) * x4a))
+ ((146.502 * f17) * x4b)) + ((559.829 * f22) * x5b))
+ ((-1547.81 * f22) * q4))

Subject To:

comp(1)  ((x1a + x1b) + x1c) ==1
comp(2)  ((x2a + x2b) + x2c) ==1
comp(3)  ((x3a + x3b) + x3c) ==1
comp(4)  ((x4a + x4b) + x4c) ==1
comp(5)  (x5b + x5c) ==1
cmbP(1)  ((((((f5 * x1a) + (f8 * x2a))
- (f15 * x3a)) - (f12 * x3a)) - ((0.412 * y1) * x3a))
+ (f11 + f19)) <=0
cmbP(2)  (((((((f5 * x1b) + (f8 * x2b))
- (f15 * x3b)) - (f12 * x3b)) + ((0.412 * y1) * x3a))
- ((0.055 * y1) * x3b)) + (0 * f2)) <=0
cmbP(3)  ((((((f5 * x1c) + (f8 * x2c))
- (f15 * x3c)) - (f12 * x3c)) + ((0.055 * y1) * x3b))
+ (0 * f2)) <=0
cmbP(4)  (((((((f4 * x1a) + (f13 * x3a))
- (f10 * x2a)) - (f8 * x2a)) - (f9 * x2a))
```

```
- ((0.412 * y2) * x2a)) + (f7 + f20)) <=0
cmbP(5)  (((((((f4 * x1b) + (f13 * x3b))
- (f10 * x2b)) - (f8 * x2b)) - (f9 * x2b))
+ ((0.412 * y2) * x2a)) - ((0.055 * y2) * x2b))
+ (0 * f2)) <=0
cmbP(6)  (((((((f4 * x1c) + (f13 * x3c))
- (f10 * x2c)) - (f8 * x2c)) - (f9 * x2c))
+ ((0.055 * y2) * x2b)) + (0 * f2)) <=0
cmbP(7)  ((((((f10 * x2a) + (f14 * x3a))
- (f3 * x1a)) - (f6 * x1a)) - ((0.412 * y3) * x1a))
+ (f2 + f21)) <=0
cmbP(8)  (((((((f10 * x2b) + (f14 * x3b))
- (f3 * x1b)) - (f6 * x1b)) + ((0.412 * y3) * x1a))
- ((0.055 * y3) * x1b)) + (0 * f2)) <=0
cmbP(9)  ((((((f10 * x2c) + (f14 * x3c))
- (f3 * x1c)) - (f6 * x1c)) + ((0.055 * y3) * x1b))
+ (0 * f2)) <=0
cmbP(10)  (((((f15 * x3a) + (f9 * x2a))
+ (f6 * x1a)) - (f17 * x4a)) + (0 * f2)) <=0
cmbP(11)  (((((f15 * x3b) + (f9 * x2b))
+ (f6 * x1b)) - (f17 * x4b)) + (0 * f2)) <=0
cmbP(12)  (((((f15 * x3c) + (f9 * x2c))
+ (f6 * x1c)) - (f17 * x4c)) + (0 * f2)) <=0
cmbP(13)  ((f17 * x4a) + -(f18)) <=0
cmbP(14)  (((f17 * x4b) - (f22 * x5b)) + (0 * f2)) <=0
cmbP(15)  (((f17 * x4c) - (f22 * x5c)) + (0 * f2)) <=0
cmbP(16)  ((f22 * x5b) + -(f23)) <=0
cmbP(17)  ((f22 * x5c) + -(f24)) <=0
cmbP(18)  (((((-0.412 * y1) * x3a)
- ((0.412 * y2) * x2a)) - ((0.412 * y3) * x1a))
+ ((f2 + f7) - f11)) <=0
cmbP(19)  (((((((0.412 * y1) * x3a)
- ((0.055 * y1) * x3b)) + ((0.412 * y2) * x2a))
- ((0.055 * y2) * x2b)) + ((0.412 * y3) * x1a))
- ((0.055 * y3) * x1b)) + -(f23)) <=0
cmbP(20)  (((((0.055 * y1) * x3b)
+ ((0.055 * y2) * x2b)) + ((0.055 * y3) * x1b))
- (f24)) <=0
cmbN(1)  ((((((-(f5) * x1a) - (f8 * x2a))
+ (f15 * x3a)) + (f12 * x3a)) + ((0.412 * y1) * x3a))
+ (-(f11) - f19)) <=0
cmbN(2)  (((((((-(f5) * x1b) - (f8 * x2b))
+ (f15 * x3b)) + (f12 * x3b)) - ((0.412 * y1) * x3a))
+ ((0.055 * y1) * x3b)) + (0 * f2)) <=0
cmbN(3)  ((((((-(f5) * x1c) - (f8 * x2c))
```

```
+ (f15 * x3c)) + (f12 * x3c)) - ((0.055 * y1) * x3b))
+ (0 * f2)) <=0
cmbN(4)  (((((((-(f4) * x1a) - (f13 * x3a))
+ (f10 * x2a)) + (f8 * x2a)) + (f9 * x2a))
+ ((0.412 * y2) * x2a)) + (-(f7) - f20)) <=0
cmbN(5)  (((((((-(f4) * x1b) - (f13 * x3b))
+ (f10 * x2b)) + (f8 * x2b)) + (f9 * x2b))
- ((0.412 * y2) * x2a)) + ((0.055 * y2) * x2b))
+ (0 * f2)) <=0
cmbN(6)  (((((((-(f4) * x1c) - (f13 * x3c))
+ (f10 * x2c)) + (f8 * x2c)) + (f9 * x2c))
- ((0.055 * y2) * x2b)) + (0 * f2)) <=0
cmbN(7)  ((((((-(f10) * x2a) - (f14 * x3a))
+ (f3 * x1a)) + (f6 * x1a)) + ((0.412 * y3) * x1a))
+ (-(f2) - f21)) <=0
cmbN(8)  (((((((-(f10) * x2b) - (f14 * x3b))
+ (f3 * x1b)) + (f6 * x1b)) - ((0.412 * y3) * x1a))
+ ((0.055 * y3) * x1b)) + (0 * f2)) <=0
cmbN(9)  ((((((-(f10) * x2c) - (f14 * x3c))
+ (f3 * x1c)) + (f6 * x1c)) - ((0.055 * y3) * x1b))
+ (0 * f2)) <=0
cmbN(10)  (((((-(f15) * x3a) - (f9 * x2a))
- (f6 * x1a)) + (f17 * x4a)) + (0 * f2)) <=0
cmbN(11)  (((((-(f15) * x3b) - (f9 * x2b))
- (f6 * x1b)) + (f17 * x4b)) + (0 * f2)) <=0
cmbN(12)  (((((-(f15) * x3c) - (f9 * x2c))
- (f6 * x1c)) + (f17 * x4c)) + (0 * f2)) <=0
cmbN(13)  ((-(f17) * x4a) + f18) <=0
cmbN(14)  (((-(f17) * x4b) + (f22 * x5b))
+ (0 * f2)) <=0
cmbN(15)  (((-(f17) * x4c) + (f22 * x5c))
+ (0 * f2)) <=0
cmbN(16)  ((-(f22) * x5b) + f23) <=0
cmbN(17)  ((-(f22) * x5c) + f24) <=0
cmbN(18)  (((((0.412 * y1) * x3a)
+ ((0.412 * y2) * x2a)) + ((0.412 * y3) * x1a))
+ ((-(f2) - f7) + f11)) <=0
cmbN(19)  (((((((-0.412 * y1) * x3a)
+ ((0.055 * y1) * x3b)) - ((0.412 * y2) * x2a))
+ ((0.055 * y2) * x2b)) - ((0.412 * y3) * x1a))
+ ((0.055 * y3) * x1b)) + f23) <=0
cmbN(20)  (((((-0.055 * y1) * x3b)
- ((0.055 * y2) * x2b)) - ((0.055 * y3) * x1b))
+ f24) <=0
ff(1)  (((((f11 + f5) + f8) + f19) - f15) - f12) =0
```

```
ff(2)  ((((((f7 + f20) + f4) + f13) - f10) - f8)
- f9) =0
ff(3)  (((((f2 + f21) + f10) + f14) - f3) - f6) =0
ff(4)  (((f15 + f9) - f6) - f17) =0
ff(5)  ((f17 - f18) - f22) =0
ff(6)  ((f22 - f23) - f24) =0
ff(7)  ((((f2 + f7) + f11) - f23) - f24) =0
ff(8)  ((f12 - f13) - f14) =0
ff(9)  ((f3 - f4) - f5) =0
ff(10) ((f18 - f19) - f20) =0
fq4P   (q4 + -((x5b ^ 2))) <=0
fq4N   (-(q4) + (x5b ^ 2)) <=0

Bounds

  0 <= f2  <= 1000,   0 <= f3  <=  500,
  0 <= f4  <=  500,   0 <= f5  <=  500,
  0 <= f6  <=  500,   0 <= f7  <= 1000,
  0 <= f8  <=  500,   0 <= f9  <=  500,
  0 <= f10 <=  500,   0 <= f11 <=  500,
  0 <= f12 <=  500,   0 <= f13 <=  500,
  0 <= f14 <=  500,   0 <= f15 <=  500,
  0 <= f17 <=  500,   0 <= f18 <=  500,
  0 <= f19 <=  500,   0 <= f20 <=  500,
  0 <= f21 <=  500,   0 <= f22 <=  500,
 75 <= f23 <=   75,   0 <= f24 <=  500,
  0 <= x1a <=    1,   0 <= x1b <=    1,
  0 <= x1c <=    1,   0 <= x2a <=    1,
  0 <= x2b <=    1,   0 <= x2c <=    1,
  0 <= x3a <=    1,   0 <= x3b <=    1,
  0 <= x3c <=    1,   0 <= x4a <=    1,
  0 <= x4b <=    1,   0 <= x4c <=    1,
  0 <= x5b <=    1,   0 <= x5c <=    1,
  0 <= y1  <=  500,   0 <= y2  <=  500,
  0 <= y3  <= 500,
  -1e+06 <= q4 <=   1e+06
```

- Example 2 of Section 5
  This problem belongs to the class of *quadratic programming* and it corresponds to **case 5** of Problem 2.7 in [16]. Specifically, for this example the following parameters are used:

$$\lambda_i \;=\; i, \quad \alpha_i \;=\; 2$$

The 10×20 A matrix, and vector b are defined as follows:

$$b \;=\; (-5, 2, -1, -3, 5, 4, -1, 0, 9, 40)^T$$

$$A = \begin{pmatrix}
-3 & 7 & 0 & -5 & 1 & 1 & 0 & 2 & -1 & 1 \\
7 & 0 & -5 & 1 & 1 & 0 & 2 & -1 & -1 & 1 \\
0 & -5 & 1 & 1 & 0 & 2 & -1 & -1 & -9 & 1 \\
-5 & 1 & 1 & 0 & 2 & -1 & -1 & -9 & 3 & 1 \\
1 & 1 & 0 & 2 & -1 & -1 & -9 & 3 & 5 & 1 \\
1 & 0 & 2 & -1 & -1 & -9 & 3 & 5 & 0 & 1 \\
0 & 2 & -1 & -1 & -9 & 3 & 5 & 0 & 0 & 1 \\
2 & -1 & -1 & -9 & 3 & 5 & 0 & 0 & 1 & 1 \\
-1 & -1 & -9 & 3 & 5 & 0 & 0 & 1 & 7 & 1 \\
-1 & -9 & 3 & 5 & 0 & 0 & 1 & 7 & -7 & 1 \\
-9 & 3 & 5 & 0 & 0 & 1 & 7 & -7 & -4 & 1 \\
3 & 5 & 0 & 0 & 1 & 7 & -7 & -4 & -6 & 1 \\
5 & 0 & 0 & 1 & 7 & -7 & -4 & -6 & -3 & 1 \\
0 & 0 & 1 & 7 & -7 & -4 & -6 & -3 & 7 & 1 \\
0 & 1 & 7 & -7 & -4 & -6 & -3 & 7 & 0 & 1 \\
1 & 7 & -7 & -4 & -6 & -3 & 7 & 0 & -5 & 1 \\
7 & -7 & -4 & -6 & -3 & 7 & 0 & -5 & 1 & 1 \\
-7 & -4 & -6 & -3 & 7 & 0 & -5 & 1 & 1 & 1 \\
-4 & -6 & -3 & 7 & 0 & -5 & 1 & 1 & 0 & 1 \\
-6 & -3 & 7 & 0 & -5 & 1 & 1 & 0 & 2 & 1
\end{pmatrix}$$

- Example 3 of Section 5
  This is problem 9.2 of [16]. It involves the *Van der Vusse* reactions:

$$A \to B$$
$$B \to C$$
$$2A \to C$$

A superstructure involving two reactors and various possible recycle is postulated, and an economic objective is optimized. The precise mathematical formulation used is:

```
Minimize
  fn_n = (((((-0.001 * y1) * x3a)
+ ((0.0001 * y1) * x3b))
- ((0.001 * y2) * x4a)) + ((0.0001 * y2) * x4b))

Subject To:

lf1  ((1 * f2) + (1 * f3)) ==1
lf2  (((1 * f2) + (1 * f5)) - (1 * f9)) ==0
lf3  ((((1 * f3) + (1 * f4)) + (1 * f6))
```

```
- (1 * f21)) ==0
fg1  ((((f5 * x4a) - (f9 * x3a)) + (y1 * r1))
+ (0 * f2)) <=-5.8
fg2  ((((-(f5) * x4a) + (f9 * x3a)) - (y1 * r1))
+ (0 * f2)) <=5.8
fg3  ((((f5 * x4b) - (f9 * x3b)) + (y1 * r2))
+ (0 * f2)) <=0
fg4  ((((-(f5) * x4b) + (f9 * x3b)) - (y1 * r2))
+ (0 * f2)) <=0
fg5  ((((f5 * x4c) - (f9 * x3c)) + (y1 * r3))
+ (0 * f2)) <=0
fg6  ((((-(f5) * x4c) + (f9 * x3c)) - (y1 * r3))
+ (0 * f2)) <=0
fg7  ((((f5 * x4d) - (f9 * x3d)) + (y1 * r4))
+ (0 * f2)) <=0
fg8  ((((-(f5) * x4d) + (f9 * x3d)) - (y1 * r4))
+ (0 * f2)) <=0
fh1  (((((f4 * x3a) + (f6 * x4a)) - (f21 * x4a))
+ (y2 * r5)) + (0 * f2)) <=-5.8
fh2  (((((-(f4) * x3a) - (f6 * x4a)) + (f21 * x4a))
- (y2 * r5)) + (0 * f2)) <=5.8
fh3  (((((f4 * x3b) + (f6 * x4b)) - (f21 * x4b))
+ (y2 * r6)) + (0 * f2)) <=0
fh4  (((((-(f4) * x3b) - (f6 * x4b)) + (f21 * x4b))
- (y2 * r6)) + (0 * f2)) <=0
fh5  (((((f4 * x3c) + (f6 * x4c)) - (f21 * x4c))
+ (y2 * r7)) + (0 * f2)) <=0
fh6  (((((-(f4) * x3c) - (f6 * x4c)) + (f21 * x4c))
- (y2 * r7)) + (0 * f2)) <=0
fh7  (((((f4 * x3c) + (f6 * x4d)) - (f21 * x4d))
+ (y2 * r8)) + (0 * f2)) <=0
fh8  (((((-(f4) * x3c) - (f6 * x4d)) + (f21 * x4d))
- (y2 * r8)) + (0 * f2)) <=0
fm1  (-(x5a) + ((y1 * r1) + (y2 * r5))) <=-5.8
fm2  (x5a + ((-(y1) * r1) - (y2 * r5))) <=5.8
fm3  (-(x5c) + ((y1 * r3) + (y2 * r7))) <=0
fm4  (x5c + ((-(y1) * r3) - (y2 * r7))) <=0
fm5  (-(x5d) + ((y1 * r4) + (y2 * r8))) <=0
fm6  (x5d + ((-(y1) * r4) - (y2 * r8))) <=0
rate1  ((r1 + (0.1 * x3a)) + (0.01 * (x3a ^ 2))) <=0
rate2  ((-(r1) - (0.1 * x3a))
+ (-0.01 * (x3a ^ 2))) <=0
rate3  ((r2 - (0.1 * x3a)) + (0.01 * x3b)) <=0
rate4  ((-(r2) + (0.1 * x3a)) - (0.01 * x3b)) <=0
rate5  (r3 - (0.01 * x3b)) <=0
```

```
rate6  (-(r3) + (0.01 * x3b)) <=0
rate7  (r4 + (-0.01 * (x3a ^ 2))) <=0
rate8  (-(r4) + (0.01 * (x3a ^ 2))) <=0
rate9  ((r5 + (0.1 * x4a)) + (0.01 * (x4a ^ 2))) <=0
rate10  ((-(r5) - (0.1 * x4a))
+ (-0.01 * (x4a ^ 2))) <=0
rate11  ((r6 - (0.1 * x4a)) + (0.01 * x4b)) <=0
rate12  ((-(r6) + (0.1 * x4a)) - (0.01 * x4b)) <=0
rate13  (r7 - (0.01 * x4b)) <=0
rate14  (-(r7) + (0.01 * x4b)) <=0
rate15  (r8 + (-0.01 * (x4a ^ 2))) <=0
rate16  (-(r8) + (0.01 * (x4a ^ 2))) <=0

Bounds

  0 <= f2  <=    1,   0 <= f3  <=    1,
  0 <= f4  <=    1,   0 <= f5  <=    1,
  0 <= f6  <=    1,   0 <= f9  <=    1,
  0 <= f21 <=    1,   0 <= x3a <=  5.8,
  0 <= x3b <=  5.8,   0 <= x3c <=  5.8,
  0 <= x3d <=  5.8,   0 <= x4a <=  5.8,
  0 <= x4b <=  5.8,   0 <= x4c <=  5.8,
  0 <= x4d <=  5.8,   0 <= x5a <=  5.8,
  0 <= x5b <=  5.8,   0 <= x5c <=  5.8,
  0 <= x5d <=  5.8,
-10 <= r1  <=   10, -10 <= r2  <=   10,
-10 <= r3  <=   10, -10 <= r4  <=   10,
-10 <= r5  <=   10, -10 <= r6  <=   10,
-10 <= r7  <=   10, -10 <= r8  <=   10,
  0 <= y1 <=   40 0 <= y2 <=   40,
```

- Example 4 of Section 5.
  This last example is a variation of the first one with a more complicated supestructure. We allow for 4 reactors (in parallel and in series) with various recycle streams. The precise mathematical formulation is as follows:

```
Minimize
  obj = (((((((147.62 * f14) * x11a)
+ ((-445.544 * f14) * x11b))
+ ((2793.79 * f15) * x12b))
+ ((686.426 * f14) * x11a))
+ ((146.502 * f14) * x11b))
+ ((559.829 * f15) * x12b))
+ ((-1547.81 * f15) * q4))
```

```
Subject To:

comp(1)  ((x10a + x10b) + x10c) ==  1
comp(2)  ((x3a + x3b) + x3c) ==1
comp(3)  ((x6a + x6b) + x6c) ==1
comp(4)  ((x8a + x8b) + x8c) ==1
comp(5)  (x12b + x12c) ==1
comp(6)  ((x11a + x11b) + x11c) ==1
cmbP(1)  (((((f5 * x10a) - (f7 * x3a))
- (f4 * x3a)) - ((0.412 * y1) * x3a))
+ (f2 + f11)) <=0
cmbP(2)  ((((((f5 * x10b) - (f7 * x3b))
- (f4 * x3b)) + ((0.412 * y1) * x3a))
- ((0.055 * y1) * x3b)) + (0 * f2)) <=0
cmbP(3)  (((((f5 * x10c) - (f7 * x3c))
- (f4 * x3c)) + ((0.055 * y1) * x3b))
+ (0 * f2)) <=0
cmbP(4)  ((((f4 * x3a) - (f23 * x6a))
- ((0.412 * y3a) * x6a)) + (f3 + f26)) <=0
cmbP(5)  (((((f4 * x3b) - (f23 * x6b))
+ ((0.412 * y3a) * x6a)) - ((0.055 * y3a) * x6b))
+ (0 * f2)) <=0
cmbP(6)  ((((f4 * x3c) - (f23 * x6c))
+ ((0.055 * y3a) * x6b)) + (0 * f2)) <=0
cmbP(7)  (((((f6 * x10a) + (f23 * x6a))
- (f24 * x8a)) - ((0.412 * y3b) * x8a)) + f27) <=0
cmbP(8)  ((((((f6 * x10b) + (f23 * x6b))
- (f24 * x8b)) + ((0.412 * y3b) * x8a))
- ((0.055 * y3b) * x8b)) + (0 * f2)) <=0
cmbP(9)  (((((f6 * x10b) + (f23 * x6c))
- (f24 * x8c)) + ((0.055 * y3b) * x8b))
+ (0 * f2)) <=0
cmbP(10)  ((((f24 * x8a) - (f10 * x10a))
- ((0.412 * y3c) * x10a)) + f28) <=0
cmbP(11)  (((((f24 * x8b) - (f10 * x10b))
+ ((0.412 * y3c) * x10a)) - ( (0.055 * y3c) * x10b))
+ (0 * f2)) <=0
cmbP(12)  ((((f24 * x8c) - (f10 * x10c))
+ ((0.055 * y3c) * x10b)) + (0 * f2)) <=0
cmbP(13)  ((f14 * x11a) + ((((-(f11) - f13)
- f26) - f27) - f28)) <=0
cmbP(14)  (((f14 * x11b) - (f15 * x12b))
+ (0 * f2)) <=0
cmbP(15)  (((f14 * x11c) - (f15 * x12c))
```

```
+ (0 * f2)) <=0
cmbP(16)  ((f15 * x12b) + -(f18)) <=0
cmbP(17)  ((f15 * x12c) + -(f16)) <=0
cmbP(18)  ((((((-0.412 * y1) * x3a)
- ((0.412 * y3a) * x6a)) - ((0.412 * y3b) * x8a))
- ((0.412 * y3c) * x10a)) + ((f2 + f3) - f13)) <=0
cmbP(19)  ((((((((((0.412 * y1) * x3a)
- ((0.055 * y1) * x3b)) + ((0.412 * y3a) * x6a))
- ((0.055 * y3a) * x6b)) + ((0.412 * y3b) * x8a))
- ((0.055 * y3b) * x8b)) + ((0.412 * y3c) * x10a))
- ((0.055 * y3c) * x10b)) + -(f18)) <=0
cmbP(20)  ((((((0.055 * y1) * x3b)
+ ((0.055 * y3a) * x6b)) + ((0.055 * y3b) * x8b))
+ ((0.055 * y3c) * x10b)) + -(f16)) <=0
cmbP(21)  ((((f7 * x3a) + (f8 * x10a))
- (f14 * x11a)) + (0 * f2)) <=0
cmbP(22)  ((((f7 * x3b) + (f8 * x10b))
- (f14 * x11b)) + (0 * f2)) <=0
cmbP(23)  ((((f7 * x3c) + (f8 * x10c))
- (f14 * x11c)) + (0 * f2)) <=0
cmbN(1)  (((((-(f5) * x10a) + (f7 * x3a))
+ (f4 * x3a)) + ((0.412 * y1) * x3a))
+ (-(f2) - f11)) <=0
cmbN(2)  ((((((-(f5) * x10b) + (f7 * x3b))
+ (f4 * x3b)) - ((0.412 * y1) * x3a))
+ ((0.055 * y1) * x3b)) + (0 * f2)) <=0
cmbN(3)  (((((-(f5) * x10c) + (f7 * x3c))
+ (f4 * x3c)) - ((0.055 * y1) * x3b))
+ (0 * f2)) <=0
cmbN(4)  ((((-(f4) * x3a) + (f23 * x6a))
+ ((0.412 * y3a) * x6a)) + (-( f3) - f26)) <=0
cmbN(5)  (((((-(f4) * x3b) + (f23 * x6b))
- ((0.412 * y3a) * x6a)) + ( (0.055 * y3a) * x6b))
+ (0 * f2)) <=0
cmbN(6)  ((((-(f4) * x3c) + (f23 * x6c))
- ((0.055 * y3a) * x6b)) + (0 * f2)) <=0
cmbN(7)  (((((-(f6) * x10a) - (f23 * x6a))
+ (f24 * x8a)) + ((0.412 * y3b) * x8a)) + -(f27)) <=0
cmbN(8)  ((((((-(f6) * x10b) - (f23 * x6b))
+ (f24 * x8b)) - ((0.412 * y3b) * x8a))
+ ((0.055 * y3b) * x8b)) + (0 * f2)) <=0
cmbN(9)  (((((-(f6) * x10c) - (f23 * x6c))
+ (f24 * x8c)) - ((0.055 * y3b) * x8b))
+ (0 * f2)) <=0
cmbN(10)  ((((-(f24) * x8a) + (f10 * x10a))
```

```
+ ((0.412 * y3c) * x10a)) + -(f28)) <=0
cmbN(11)  (((((-(f24) * x8b) + (f10 * x10b))
- ((0.412 * y3c) * x10a)) + ((0.055 * y3c) * x10b))
+ (0 * f2)) <=0
cmbN(12)  ((((-(f24) * x8c) + (f10 * x10c))
- ((0.055 * y3c) * x10b)) + (0 * f2)) <=0
cmbN(13)  ((-(f14) * x11a) + ((((f11 + f13)
+ f26) + f27) + f28)) <=0
cmbN(14)  (((-(f14) * x11b) + (f15 * x12b))
+ (0 * f2)) <=0
cmbN(15)  (((-(f14) * x11c) + (f15 * x12c))
+ (0 * f2)) <=0
cmbN(16)  ((-(f15) * x12b) + f18) <=0
cmbN(17)  ((-(f15) * x12c) + f16) <=0
cmbN(18)  ((((((0.412 * y1) * x3a)
+ ((0.412 * y3a) * x6a)) + ((0.412 * y3b) * x8a))
+ ((0.412 * y3c) * x10a)) + ((-(f2) - f3) + f13)) <=0
cmbN(19)  ((((((((((-0.412 * y1) * x3a)
+ ((0.055 * y1) * x3b)) - ((0.412 * y3a) * x6a))
+ ((0.055 * y3a) * x6b)) - ((0.412 * y3b) * x8a))
+ ((0.055 * y3b) * x8b)) - ((0.412 * y3c) * x10a))
+ ((0.055 * y3c) * x10b)) + f18) <=0
cmbN(20)  ((((((-0.055 * y1) * x3b)
- ((0.055 * y3a) * x6b)) - ((0.055 * y3b) * x8b))
- ((0.055 * y3c) * x10b)) + f16) <=0
cmbN(21)  ((((-(f7) * x3a) - (f8 * x10a))
+ (f14 * x11a)) + (0 * f2)) <=0
cmbN(22)  ((((-(f7) * x3b) - (f8 * x10b))
+ (f14 * x11b)) + (0 * f2)) <=0
cmbN(23)  ((((-(f7) * x3c) - (f8 * x10c))
+ (f14 * x11c)) + (0 * f2)) <=0
ff(1)  ((((f5 - f7) - f4) + f2) + f11) ==0
ff(2)  (((f3 + f26) + f4) - f23) ==0
ff(3)  ((f23 + f27) - f24) ==0
ff(4)  (((f24 + f28) - f8) - f5) ==0
ff(5)  (((((((f11 + f13) + f26) + f27)
+ f28) - f7) - f8) + f15) ==0
ff(6)  ((f15 - f18) - f16) ==0
ff(7)  ((((f2 + f3) - f13) - f18) - f16) ==0
ff(8)  ((f14 - f7) - f8) ==0
ff(9)  ((((((f3 + f4) + f6) + f26) + f27)
+ f28) - f10) ==0
ff(10)  ((((f2 + f11) + f5) - f4) - f7) ==0
ff(11)  (((f10 - f6) - f4) - f8) ==0
fq4P  (q4 + -((x12b ^ 2))) <=0
```

```
fq4N  (-(q4) + (x12b ^ 2)) <=0

Bounds

  0 <= f2  <=   500, 0  <= f3  <=  500,
  0 <= f4  <=   500, 0 <= f5  <=   500,
  0 <= f6  <=   500, 0  <= f7  <=  500,
  0 <= f8  <=   500, 0 <= f10 <=   500,
  0 <= f11 <=   500, 0  <= f13 <=  500,
  0 <= f14 <=   500, 0 <= f15 <=   500,
  0 <= f16 <=   500, 50 <= f18 <=  500,
  0 <= f23 <=   500, 0 <= f24 <=   500,
  0 <= f26 <=   500, 0  <= f27 <=  500,
  0 <= f28 <=   500,
  0 <= x3a  <=    1, 0 <= x3b  <=   1,
  0 <= x3c  <=    1, 0 <= x6a  <=   1,
  0 <= x6b  <=    1, 0 <= x6c  <=   1,
  0 <= x8a  <=    1, 0 <= x8b  <=   1,
  0 <= x8c  <=    1, 0 <= x10a <=   1,
  0 <= x10b <=    1, 0 <= x10c <=   1,
  0 <= x11a <=    1, 0 <= x11b <=   1,
  0 <= x11c <=    1, 0 <= x12b <=   1,
  0 <= x12c <=    1
  0 <= y1   <=  120, 0 <= y3a <=  100,
  0 <= y3b <= 100,   0 <= y3c <=  100,
  -1e+06 <= q4 <=   1e+06
```

REFERENCES

[1] C. Adjiman, I. Androulakis, and C. Floudas, *A global optimization method, αbb, for general twice differentiable nlp's – ii. implementation and computational results*, Computers and Chemical Engineering (in press), (1997).

[2] ———, *Global optimization of minlp problems in process synthesis*, Computers and Chemical Engineering, S21 (1997), pp. 445–450.

[3] C. Adjiman, S. Dawlling, C. Floudas, and A. Neumaier, *A global optimization method, αbb, for general twice differentiable nlp's – i. theoretical advances*, Computers and Chemical Engineering (in press), (1997).

[4] C. Adjiman and C. Floudas, *Rigorous convex underestimators for general twice–differentiable problems*, Journal of Global Optimization, 9 (1996), pp. 23–40.

[5] I. Androulakis, C. Maranas, and C. Floudas, *αbb : A global optimization method for general constrained nonconvex problems*, Journal of Global Optimization, 7 (1995), pp. 337–363.

[6] ———, *Predicting oligopeptide conformations via deterministic optimization*, Journal of Global Optimization, 11 (1997), pp. 1–34.

[7] I. Androulakis and G. Reklaitis, *Analysis of the spurious behavior of asynchronous relaxation algorithms*, Computers and Chemical Engineering, 123 (1994), pp. 456–789.

[8] I. Androulakis, V. Visweswaran, and C. Floudas, *Distributed decomposition based approaches in global optimizatio*, in State of the Art in Global Optimization: Computational Methods and Applications, C. Floudas and P. Pardalos, eds., Kluwer Academic Publishers:Book Series on Nonconvex Optimization and its Applications, 1996, pp. 285–302.

[9] D. Bertsekas and J. Tsitsiklis, *Parallel and Distributed Computing*, Prentice Hall, 1989.

[10] E. Dijkstra, W. Feijen, and A. vanGasteren, *Derivation of a termination detection algorithm for distributed computations*, Information Processing Letters, 16 (1983), pp. 217–219.

[11] E. Dijkstra and C. Scholten, *Termination detection for diffusing computations*, Information Processing Letters, 11 (1980), pp. 1–6.

[12] J. Eckstein, *Parallel Branch-and-Bound Algorithms for General Mixed Integer Programming on the CM–5*, Thinking Machines Corporation Technical Report TMC–257, 1993.

[13] C. Floudas, *Deterministic global optimization in design, control and computational chemistry*, in IMA Proceedings: Large Scale Optimization with Applications. Part II:Optimal Design and Control, L. Biegler, A. Conn, L. Coleman, and F.Santosa, eds., Springer-Verlag, 1995, pp. 129–184.

[14] ———, *Nonlinear and Mixed Integer Optimization: Fundamentals and Applications*, Oxford University Press, 1995.

[15] C. Floudas and I. Grossmann, *Algorithmic approaches to process synthesis: Logic and global optimization*, in Proceedings of FOCAPD'94, M. Doherty and L. Biegler, eds., 1995, pp. 198–221.

[16] C. Floudas and P. Pardalos, *A Collection of Test Problems for Constrained Global Optimization Algorithms*, Springer-Verlag, 1990.

[17] ———, *State of the Art in Global Optimization*, Kluwer Academic Publishers, 1996.

[18] B. Gendron and T. Crainic, *Parallel branch-and-bound algorithms: Survey and synthesis*, Operations Research, 42 (1994), pp. 1042–1066.

[19] I. Grossmann(Editor), *Global Optimization in Chemical Engineering*, Kluwer Academic Publishers, 1996.

[20] P. Laursen, *Simple approaches to parallel branch and bound*, Parallel Computing, 19 (1993), pp. 143–152.

[21] G. Li and B. Wah, *Coping with anomalies in parallel branch-and-bound algorithms*, IEEE Transactions on Computers, 35 (1986), pp. 568–573.

[22] C. Maranas and C. Floudas, *Global minimum potential energy conformations of small molecules*, Journal of GLobal Optimization, 4 (1994), pp. 135–170.

[23] P. Pardalos, A. Phillips, and J. Rosen, *Topics in Parallel Computing in Mathematical Programming*, Science Press, 1992.

[24] P. Pardalos, G. Xue, and P. Panagiotopoulos, *Parallel algorithms for global optimization: Methods and techniques*, in Solving Combinatorial Optimization Problems in Parallel, Lecture Notes in Computer Science, vol. 1054, A. Ferreira and P. Pardalos, eds., Springer-Verlag, 1996, pp. 232–247.

[25] J. Pekny and D. Miller, *A parallel branch and bound algorithm for solving large asymmetric traveling salesman problems*, Mathematical Programming, 55 (1992), pp. 17–33.

[26] M. Quinn, *Analysis and implementation of branch and bound algorithms on a hypercube multicomputer*, IEEE Transactions on Computers, 39 (1990), pp. 384–387.

[27] O. Vorneberg, *Transputer networks for operations research*, Journal of Microcomputer Applications, 13 (1990), pp. 69–79.