AIAA JOURNAL
Vol. 44, No. 9, September 2006

General Implementation of Multilevel Parallelization in a Gradient-Based Design Optimization Algorithm

S. D. Rajan*
Arizona State University, Tempe, Arizona 85287
A. D. Belegundu†
Pennsylvania State University, University Park, Pennsylvania 16802
and
A. S. Damle,‡ D. Lau,§ and J. St. Ville‖
Hawthorne & York, International, Phoenix, Arizona 85008

DOI: 10.2514/1.17470

As product designs have become more sophisticated, both the simulation models (e.g., finite element models) and the design optimization models have grown bigger. To keep pace with this increase in problem size, we present and implement an optimization strategy that can run on a computing cluster with demonstrable efficiency. First, parallelism is implemented in the context of gradient calculations using divided differences. Then, parallelism is achieved in the context of both direction-finding and line-search steps. Parallel direction finding improves the convergence rate as opposed to just cutting down the amount of arithmetic. A new algorithm based on method of feasible directions is discussed that obtains better optima and is also computationally faster. Implementation details regarding distribution of computing tasks to improve scalability and load balancing are presented. Numerical examples show the efficiency of the developed methodology on a relatively small computing cluster. Gains of about 7:1 have been obtainable using 16 processors on some test problems. Importantly, the framework presented can be developed by researchers using other gradient-based optimization codes on different computing platforms.

I. Introduction

FINITE-ELEMENT based design optimization is now a relatively well-established methodology for engineering design. The use of this methodology involves several areas and techniques such as geometric modeling, mesh generation, finite element analysis, numerical optimization techniques, etc. Advances in each of these areas have made the overall design process more versatile, especially when and where there is a tight integration between these areas. This integration enables an end-to-end solution with reduced designer intervention. Yet there remain more challenges to meet and hurdles to overcome. As finite-element models have become more sophisticated and detailed, the execution time has also increased in spite of advances in hardware technology. Compressing the design cycle time so as to reduce the time required for design and redesign process requires more advances in hardware, software, and algorithms.

The focus of this paper is to introduce 1) hardware and software issues with regard to parallel computing, which were deemed to be important in a recent conference panel, 2) techniques for parallelizing a constrained gradient-based optimization method, viz. the method of feasible directions along with results and conclusions, and 3) the software architecture used in 2), which the reader may port to other gradient optimization techniques and algorithms.

It is interesting to note that today design engineers have access to almost every single type of computer hardware and operating system. A typical engineering computing environment is increasingly varied and heterogeneous. They include 32-bit desktop systems, 64-bit workstations, cluster of 32-bit machines, cluster of 64-bit machines, 32-bit and 64-bit machines with multiple processors in a shared memory mode as well as supercomputers with several processors, shared memory and vector processing capabilities. The operating systems are also varied: Microsoft Windows, the various flavors of Linux and Unix, and others. A typical design engineer works with a 32-bit or 64-bit desktop system. This system is used primarily 1) for pre- and postprocessing, 2) for design problems that can be executed on these systems in a reasonable amount of time, and 3) as a gateway to faster computing platforms. When the design engineer needs to solve bigger problems, different types of parallel computing architectures can be used. Hardware and software issues as noted in the following sections were presented and discussed at a recent AIAA conference [1]. Parallelism to date in optimization algorithms is discussed in Section II.

Systems with multiple processors have been around a long time, especially in the form of mainframes and supercomputers. However, in the last decade there has been concerted effort to use commonly available CPUs in a variety of parallel processing architectures starting with Beowulf clusters. Today, the three most popular architectures are clusters, constellations, and massively parallel processing (MPP) systems [2]. Clusters are systems made up of several computing nodes. Each node can potentially work in a stand-alone mode. It has its own CPU and memory. However, the nodes are clustered together in a network to yield faster computing platforms. We have a Beowulf cluster when commodity machines are used as nodes and are networked using commodity networking hardware. With today's hardware reality in terms of the number of different CPU manufacturers, there really is no difference between PCs and workstations. One can look at parallelism taking place at two levels: by increasing the number of nodes in a cluster, and by increasing the number of processors at a node. There are specialized hardware manufacturers that provide computing nodes with multiple processors configured as a symmetric multiprocessor (SMP) or as distributed shared memory (DSM) machines. Hence constellations can be thought of as providing "clustering" at two levels. Finally, it is more difficult to exactly define or identify MPP systems. They could be a massive distributed memory system, or a large shared memory system employing cache coherence, or a large

Received 1 May 2005; accepted for publication 1 April 2006.

*Professor, Civil and Environmental Engineering Department.
†Professor, Mechanical and Nuclear Engineering Department.
‡Design Software Engineer.
§Manager, FEA Group.
‖President.

vector system, and so on. What is perhaps clear is that these MPP systems are much more expensive to buy, build, and maintain than commodity machines.

With the termination of the design of faster processors by at least two major chip manufacturers (at least for the time being), parallel processing and algorithm developments have obtained a renewed and higher lease on life. The implications for software developers are that 1) hardware evolution will still take place at a rapid pace for the foreseeable future but gains will come more slowly than before, and 2) they must protect their investment with software standards that should evolve transparently with the changes in hardware. We note that a list of the top supercomputer sites is given at http://www.top500.org. Clusters, where independent computers are connected, account for about 58% of total usage, whereas massively parallel machines account for 25%.

In comparison to the development of (faster) hardware, little effort has been invested in the development and ready availability of software tools. Two widely used approaches in parallelizing computations involve the use of threads (usually on shared memory systems) and some form of message-passing (usually on distributed memory systems). There have been successful efforts at using both threads in the form of OpenMP directives and message-passing in the form of MPI calls to create high-performance software systems. OpenMP is used primarily with shared memory systems whereas MPI is the choice for distributed memory systems. MPI implementations use TCP/IP protocol or with custom switching hardware, proprietary protocol such as Myrinet, etc., for increased bandwidth. There are two primary software-related issues.

Development and maintenance cost: The development and maintenance cost of any software system is a function of a number of parameters. In any case, there are two parameters that software developers need to pay particular attention to. First is the choice of programming language. Second is the use of libraries (or function calls) for interprocess communication. Given portability considerations at the top of the list, the language of choice is either FORTRAN or C/C++. It appears that MPI and OpenMP are increasingly the libraries of choice for interprocess communication.

Load balancing and scalability: Any parallel algorithm needs to address the issues of load balancing and scalability. It is a challenge to develop an algorithm where a task can be split into n equally compute intensive parallel subtasks. One can have a fine granularity, where many processes work on the task that needs to be done (perhaps making load balancing easier), or coarse granularity that allows a larger ratio of computation to communication. The optimal granularity depends on hardware architecture, the number of processors, the problem size, and the specifics of the problem being solved. On the other hand, scalability is the ability to use increasing number of processors as efficiently as possible. For most algorithms, a saturation point is reached with increasing number of processors beyond which the speedup drops. In other words, there is a nonlinear relationship between speedup and the number of processors. Finding the optimal number of processors for a given task is a challenging scheduling problem.

The software development challenge is to design a system that will scale with available hardware. To make the most of the system, one should know the topology of the system and the desired topology for the application's threads and/or processes. Software development under this scenario offers challenges that are unique to parallel computations. There has been considerable attention paid to coarse-grain, single-level parallelization of optimization algorithms [3–8]. In this paper we discuss multilevel (two-level) parallelism involving only the components of design optimization (DO) algorithm. Further benefits can be accrued by also carrying out function evaluation [finite element analysis (FEA)] in parallel. We discuss this three-level parallelization in a companion paper, currently under preparation.

To date, parallel computing in optimization has been mainly used to reduce arithmetic. Prime examples include parallel fitness evaluations of a population of designs in genetic algorithms [9], forward difference computation of derivatives in engineering optimization [3,8], in constructing response surfaces that are then used for optimization [4,7], and in solution of matrix equations that arise in simplex routines in linear programming. Coarse-grained parallelism has been used by many researchers, such as parallel execution of nonconvex problems from different starting points, asynchronous search [10], parallel nongradient pattern search, etc. Some work has been done on parallelizing simulated annealing algorithms for discrete and global optimization [11].

Parallel algorithms for gradient-based nonlinear programming are mainly devoted to unconstrained optimization, particularly quasi-Newton methods, where search directions have been selected from a single-parameter family of Hessians [12]. This idea has been generalized to other techniques such as successive quadratic programming (SQP) [13], and in feasible direction methods [14]. Parallel algorithms have also been used for unconstrained one-dimensional search [15], parallel equal interval search for zero-finding within the feasible directions procedure [16] and parallel finite elements with optimization [17].

Publicly available parallel optimization strategies have been cited in OPTIX software at University of Siegen [18], DAKOTA software at Sandia Labs [19], TAO at Argonne Labs [20], and several others. Single processor based optimization of large nonlinear programming problems has also been the focus in the Lancelot package [21]. Virtually every simulation software has some optimization capability (e.g., ANSYS, NASTRAN, LMS, iSIGHT, GENESIS, and ASTROS, to name a few), and some parallelism exists in these with regard to carrying out the simulations ("function calls").

Design optimization involves the modification of structural dimensions, cross section parameters, thicknesses, shape parameters, material parameters, etc. The objective function is related to weight, cost, noise levels, or multiple attributes. Constraints relate to stress, resonances, or stability, to name a few. This class of problems is characterized by the presence of a large number of nonconvex constraints. The nonlinear programming problem is of the form

$$\langle \text{minimize } f(\mathbf{x}) \text{: subject to } g_j(\mathbf{x}) \leq 0 \text{ and } \mathbf{x}^L \leq \mathbf{x} \leq \mathbf{x}^U \rangle \quad (1)$$

Nonlinear equality constraints are not usually present in design optimization. The terms "analysis" and "function call" used in this paper mean the same thing: an evaluation of the objective and constraint functions in the nonlinear programming problem. A typical analysis, say, static analysis using finite elements, involves solving a system of equations

$$\mathbf{K}_{n\times n}\mathbf{D}_{n\times 1} = \mathbf{F}_{n\times 1} \quad (2)$$

where $\mathbf{K}$ is the system stiffness matrix, $\mathbf{D}$ is the vector of nodal displacements, $\mathbf{F}$ is the vector of nodal forces, and n is the number of degrees of freedom. Because finite element analyses are computationally expensive, and because of the presence of a large number of variables, gradient-based methods are still the workhorse for structural optimization. Among gradient-based methods, method of feasible directions (used in this paper), generalized reduced gradient, and sequential quadratic programming are the most popular.

II. A Parallel Computing Software System for Design Optimization

A software system, HYI-3D [22], has been developed using object-oriented concepts in C++ and uses MPI for message passing. The software suite is a collection of independent modules that can be plugged together to create the required set of finite element analysis and design optimization capabilities and can be executed either sequentially or in parallel. Algorithmic details used in the software system are now presented. The details of the multilevel parallel implementation are discussed in Section III.

A. Parallel Gradient Evaluation

In HYI-3D, gradients are evaluated using the forward difference method: derivative of a function $f(\mathbf{x})$, which can represent an objective or a constraint function, with respect to the ith design variable is approximated by

$$\frac{\partial f(x_1, x_2, \ldots x_n)}{\partial x_i} = \frac{f(x_1, x_2, \ldots x_i + \Delta x_i, \ldots x_n) - f(x_1, x_2, \ldots x_n)}{\Delta x_i} \quad (3)$$

The number of FEAs required during gradient evaluation (GE) is equal to the number of design variables. When multiple processors are available, gradient evaluation can be parallelized such that the number of FEAs is divided equally among the available processes.

B. Parallel Direction Finding: A New Algorithm for Method of Feasible Directions

Feasible direction method (FDM) was first developed by Zoutendijk [23]. Two essential steps are involved in this method: 1) direction finding and 2) constrained line search. In direction finding, a linear or quadratic program (LP or QP) is solved at the current point. Constraint curvature is a main source of difficulty in the algorithm. Pushoff factors [23] are traditionally used to deflect away from nonlinear constraints, as repeatedly hitting the boundaries causes the step sizes to become small, which is called a "jamming" phenomenon. Alternatively, large constraint pushoffs will result in minimal reductions in objective function, or "zig-zagging." Note that even when constraint gradients are normalized, default pushoff factors of unity are ineffective in many problems. Yet another difficulty relates to an inactive constraint abruptly becoming active (relates to difficulty in choosing a constraint thickness parameter). These are illustrated in Fig. 1. Of course, if a constraint is linear, then a zero pushoff value, allowing a tangent direction (at least with respect to that particular constraint), is best. To illustrate this, a very simple example is presented here that the reader can recreate:

$$\begin{array}{ll} \text{minimize} & f = -x_2 \\ \text{subject to} & g \equiv x_1^2 - x_1 + x_2 - 1 \le 0 \\ & -1.5 \le x_1, x_2 \le 1.5 \end{array} \quad (4)$$

with a starting point of $\mathbf{x} = [-0.6, 0]^T$. Figures 1a and 1b illustrate jamming and zig-zagging, respectively.

To alleviate the difficulty in choosing pushoff factors for various problems, we suggest a strategy which shows promise. At a conceptual level, the idea is stated as follows. Gradient algorithms, including MFD, can be denoted as

$$\mathbf{x}^{k+1} = \boldsymbol{A}[\mathbf{x}^k, \mathbf{d}^k(\gamma_k)] \quad (5)$$

where γ_k is a parameter in the direction-finding subproblem, and $\boldsymbol{A}$ represents a mapping. Invariably, γ_k balances objective function reduction with constraint satisfaction. Most gradient methods have an implicit value set for this parameter, such as unity, which works relatively well for well-scaled, small-sized problems. Parallel computers can be used to exploit multiple search directions to improve robustness and/or convergence. One or two past research efforts have attempted parametrizing a family of search directions but with little success (as noted by the researchers themselves). Recently [14], the Hessian matrix $\mathbf{Q}$ is replaced with several matrices of the form $\gamma_p \mathbf{Q}$ with γ_p being a set of positive random numbers. Of the several search directions, the algorithm chooses the best one after performing line searches. To summarize the concept, we present here a one-parameter family of search directions as denoted conceptually by Eq. (5).

Standard direction: First, the standard search direction $\mathbf{d} = \mathbf{d}_{\text{std}}$ is obtained from solving the LP:

$$\begin{array}{lll} \text{minimize} & \beta & \\ \text{subject to} & \nabla f^T \mathbf{d} \le \beta & \\ & \nabla g_j^T \mathbf{d} \le \beta & j \in J_{\text{NL}} \\ & \nabla g_j^T \mathbf{d} \le -g_j^0 & j \in J_B \\ & -1 \le d_i \le 1 & i = 1, \ldots, \text{NDV} \end{array} \quad (6)$$

where J_{NL} is the set of active nonlinear constraints and can be defined as $J_{\text{NL}} = \{j\colon g_j + \varepsilon \ge 0, j = 1, \ldots, m\}$ (ε is active constraint thickness and m is the number of constraints), J_B is the set of active bound constraints and can be defined as $J_B = \{j\colon g_{jB} + \varepsilon_B \ge 0, j = 1, \ldots, \text{NDV}\}$ (ε_B is the bound constraint thickness), and NDV is the number of design variables.

Tangent direction: Second, a tangent search direction $\mathbf{d} = \mathbf{d}_t$ is obtained from the LP:

$$\begin{array}{lll} \text{minimize} & \beta & \\ \text{subject to} & \nabla f^T \mathbf{d} \le \beta & \\ & \nabla g_j^{\mathrm{T}} \mathbf{d} \le \varepsilon & j \in J_{\text{NL}} \\ & \nabla g_j^{\mathrm{T}} \mathbf{d} \le -g_j^0 & j \in J_B \\ & -1 \le d_i \le 1 & i = 1, \ldots, \text{NDV} \end{array} \quad (7)$$

where ε is a negative real number close to zero.

γ-directions: Upon obtaining $\mathbf{d}_t$ and $\mathbf{d}_{\text{std}}$, search directions $\mathbf{d} = \mathbf{d}_{\gamma i}$, $i = 1, \ldots, n_\gamma$ are obtained from the "relaxed LP subproblems":

$$\begin{array}{lll} \text{minimize} & \beta & \\ \text{subject to} & \nabla f^{\mathrm{T}} \mathbf{d} \le \gamma\, (\nabla f^{\mathrm{T}} \mathbf{d}_t) & 0 < \gamma < 1 \\ & \nabla g_j^{\mathrm{T}} \mathbf{d} \le \beta & j \in J_{\text{NL}} \\ & \nabla g_j^{\mathrm{T}} \mathbf{d} \le -g_j^0 & j \in J_B \\ & -1 \le d_i \le 1 & i = 1, \ldots, \text{NDV} \end{array} \quad (8)$$

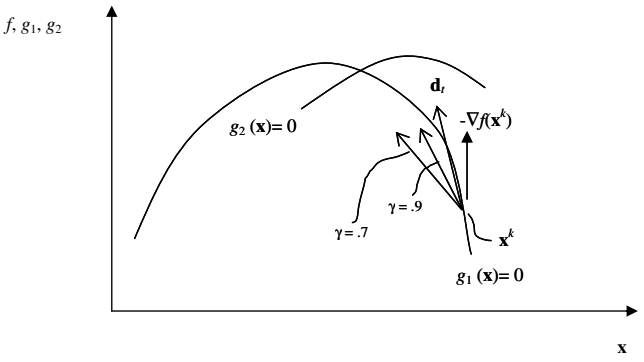


Fig. 2 γ-parameter use in computing search directions.

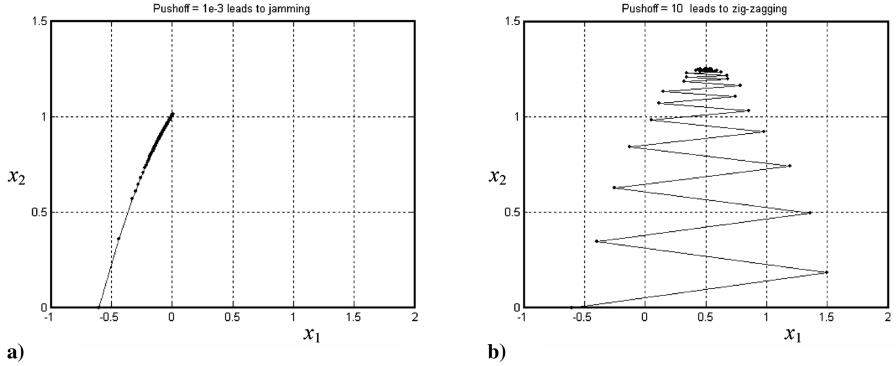
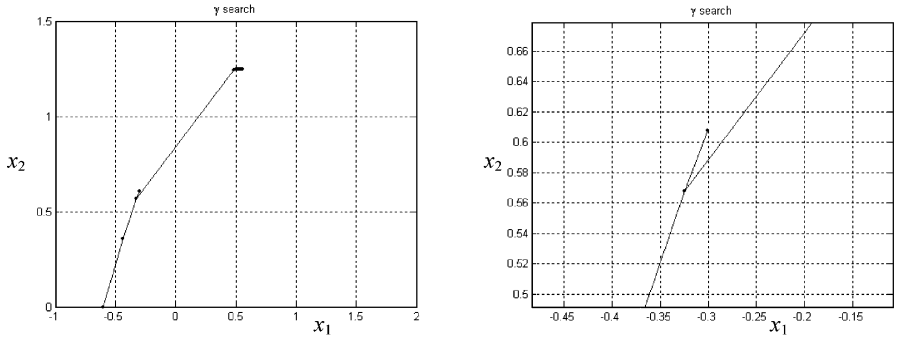


Fig. 1 a) Pushoff $\theta = 0.001$ (small), b) pushoff $\theta = 10$ (large).



a) b)

Fig. 3 a) Example in Eq. (4) solved using γ-search, b) closeup view of trial directions used at a point.

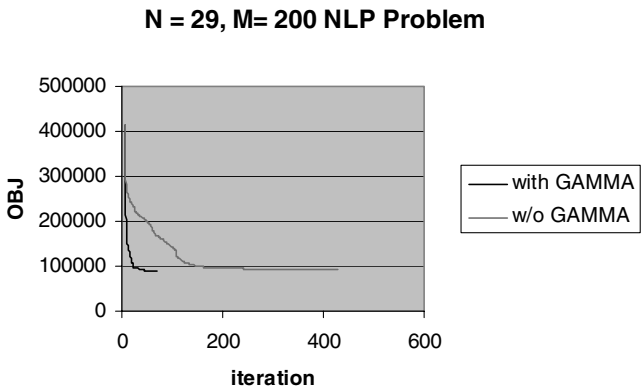


Fig. 4 Accelerated convergence on a truss optimization problem using γ-search.

Essentially, each direction is in the usable-feasible cone A–B depicted in Fig. 2. Solution of the first two LPs helps generate a set of reasonable γs. Redoing the example in Eq. (4) using this idea, we obtain the solution path shown in Fig. 3.

Effectiveness (robustness and efficiency) of γ-search on a larger optimization truss sizing problem is shown in Fig. 4. It is noted that γ-search is not better than using a single pushoff factor on all problems. But it is generally at least as good, and parallel computation makes it computationally efficient.

C. Parallel Line Search

This particular line search (LS) algorithm based on Avriel search [16] has been embedded within MFD [22]. We present a few details for completeness, so that the reader can better appreciate the discussion on parallel implementation in Section III.

$$\alpha_k = \arg\ \min\ \{f(\alpha), \text{subject to } g_i(\alpha) \le 0, i = 1, \ldots, m\} \qquad (9)$$

where $f(\alpha) \equiv f(\mathbf{x}^k + \alpha \mathbf{d}^k)$, $\mathbf{x}^k$ = current point, and $\mathbf{d}^k$ = direction vector. Steps 1 and 2, described next, aim at determining maximum step size to the boundary as defined by

$$\alpha_k = \arg\ \min\ \{\alpha\text{: subject to } g_i(\alpha) \le 0, i = 1, \ldots, m\} \qquad (10)$$

Step 1: Bracketing the zero, i.e., obtaining a bracket containing $\alpha_{\max}$

a) Initiate a march with ℓ steps (ℓ = number of processes).

b) Each process performs a function evaluation at its respective step and sends its f and $g_{\max}$ values to the master process. Note: $g_{\max} = \max\{g_i, i = 1, \ldots, m\}$.

c) The bracket is the interval whose end points have $g_{\max}$ values of opposite sign.

Step 2: Zero-finding (*equal interval sectioning*)

a) Divide the bracket found in Step 1 into $\ell + 1$ equal parts resulting in ℓ step sizes within the interval. Let these step sizes be termed interior points.

b) Each processor evaluates $g_{\max}$ at its assigned interior point.

c) These values are sent to the master processor and assembled in order.

d) The new interval is the one whose end points have $g_{\max}$ values of opposite sign and the updated step size α_U is the feasible (lesser) endpoint.

e) This process is repeated until the interval is sufficiently small in size. The interval containing the minimum of f is $[0, \alpha_U]$, also referred to as the interval of uncertainty.

Step 3: Function minimization using parallel Avriel search

The problem is to determine

$$\alpha_k = \arg\min\{f(\alpha)\text{: } 0 \le \alpha \le \alpha_U\} \qquad (11)$$

a) Obtain $2k = \ell$ points within the interval of uncertainty using the following steps. The initial interval is given by

$$d_B^0 = b - a (a \text{ and } b \text{ are the end points of the interval})$$

First block search for $i = 1$

$$\alpha_1^1 = a \qquad \alpha_2^1 = a + (1/\tau_k) d_B^0 \quad \text{where } \tau_k = \frac{1}{2}(k + \sqrt{k^2 + 4k})$$
$$\alpha_{j+1}^1 = \alpha_{j-1}^1 + (1/k) d_B^0 \quad \text{where } j = 2, 3, \ldots, 2k \qquad (12)$$

b) Compute function value at $f(\alpha_j^1)$ in parallel ($j = 2, 3, \ldots, 2k$) and obtain the value of α_j^1 which gives the minimum value of f, say $\alpha_j^1 = \alpha_I^1$.

c) Set

$$r_I = \alpha_{I-1}^1 \quad \text{and} \quad R_I = \alpha_{I+1}^1 \qquad (13)$$

Thus new reduced interval $d_B^1 = R_I - r_I = d_B^0 / n_p$. The subsequent ith block iteration (for $i \ge 2$) is given as follows:

$$\alpha_1^i = r_{i-1} \qquad \alpha_2^i = r_{i-1} + (1/\tau_k)^i d_B^0$$
$$\alpha_{j+1}^i = \alpha_{j-1}^i + (d_B^0/n_p)(\tau_k)^{1-i} \quad \text{where } (j = 2, 3, \ldots .2k) \qquad (14)$$

d) Compute function value $f(\alpha_j^i)$ in parallel. Repeat the process until the size of the final uncertainty interval falls below a predefined tolerance value.

III. Implementation Details of Multilevel Parallelism

As we see from these discussions, almost every step in the design optimization algorithm can potentially be parallelized. In this section we discuss the implementation scheme for the various parallelisms possible during design optimization. To contrast the differences between a typical sequential DO algorithm and the parallel version, we first look at a typical sequential algorithm. The algorithm has eight steps:

1) Carry out a function evaluation with the initial guess.
2) Start design iterations.
3) Carry out gradient evaluation at current design point.
4) Solve direction-finding problem.
5) Find optimal step length from line-search problem.

6) Compute the next design point.

7) Converged solution? If "no," go to step 3.

8) If "yes," carry out the final function evaluation with the optimal values.

The computationally expensive parts of the algorithm are steps involving function evaluations: steps 1, 3, 5, and 8. Step 4 also can be expensive depending on the nature of the subproblem solved to obtain the search direction. The challenge is to reduce these computationally expensive steps through parallelization.

We first introduce the nomenclature and then present the details of the implementation. In this paper, we only discuss parallelization of optimization (MFD) algorithm without nesting this with parallelization of the function evaluations (i.e., FEAs). It should be noted that parallelization involving FEAs shows promise for such a nested, three-level approach on design optimization problems involving very large FE models. This is currently being pursued.

A strict master-slave implementation is not implemented. Three types of master processors are defined whose job is to coordinate the activities and participate in the computations during gradient evaluation (GE), direction finding (DF), and line search (LS) computations and consolidate the final answer. The GE manager is the master processor during the parallel gradient computations and constructs the final gradient vectors of the objective function and the active constraints. Similarly, there are DF and LS managers.

NDV is the number of design variables.

NFV_{GE} is the number of function evaluations in GE.

NFV_{LS} is the number of function evaluations in LS.

NFV_{Total} is the total number of function evaluations (GE + LS + external).

d is the total number of search directions.

ℓ is the number of processors available for parallel LS.

f is the number of domains the FEA model is split into (also the number of processors involved in parallel FEA).

PE is the total number of processors.

P_i is the processor i, $1 \leq i \leq$ PE.

A. One-Level Parallelism

Figure 5 shows examples of one-level parallelization. We discuss four broad cases under this category next:

GE manager: Fig. 5a shows parallel GE implementation. This is an embarrassingly parallel problem and leads to good speedup when GE requires most of the DO execution time. Perfect load balancing is achieved when the number of design variables is a multiple of the number of processors. Otherwise, some processors will remain idle while others carry out one more function evaluation in the final stage. Consider the scenario shown in Fig. 5a involving four processors and 40 design variables. First, all the processors construct the list of active constraints. Then each one computes components of the objective function gradient and the active constraint gradients for their assigned design variables. Finally, GE manager (processor P1) collects and assembles these gradient components to form the objective function gradient vector and active constraint gradient matrix.

DF manager: Consider the scenario shown in Fig. 5b where search directions are computed in parallel. Each direction involves solution of a bounded LP by the revised simplex procedure. Parallel direction finding is carried out in two stages. In the first stage, two processors calculate the standard and tangent directions. In the second stage, each available processor calculates one of the remaining directions simultaneously. The number of directions to be used is a user-specified input, usually between two and four. With respect to the scenario in Fig. 5b, during stage 1, P1 and P2 calculate $\mathbf{d}_{\text{std}}$ and $\mathbf{d}_t$ respectively. In Stage 2, P1 and P2 calculate $\mathbf{d}_{\gamma 1}$ and $\mathbf{d}_{\gamma 2}$ respectively. Because $\mathbf{d}_{\text{std}}$ and $\mathbf{d}_t$ have to be computed *before* $\mathbf{d}_{\gamma 1}$ and $\mathbf{d}_{\gamma 2}$, P3 and P4 remain idle during this time. Finally, P1 collects all the direction vectors and stores them for later use during the line-search process.

LS manager (with parallel DF, serial LS): Consider the scenario shown in Fig. 5c where direction finding is carried out in parallel. Processors P1–P4 perform simultaneous line search along all the four search directions (each processor is assigned to each direction) to find optimum step size and corresponding objective function value. LS manager P1 collects all the optimum step sizes and determines the best new point based on the best objective function value.

LS manager (with parallel LS, serial DF): Finally, consider the scenario shown in Fig. 5d where line search is carried out in parallel along a specified direction. Processors P1–P4 perform line search along one direction at a time.

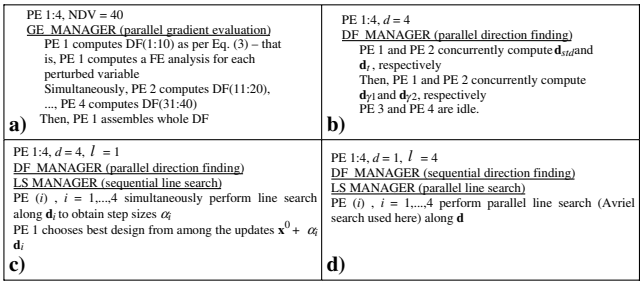


Fig. 5 Example scenarios depicting single-level parallelism.

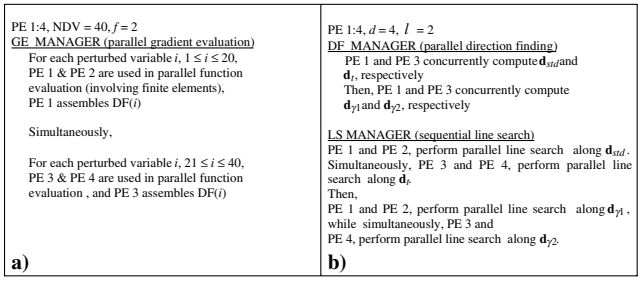


Fig. 6 Example scenarios depicting two-level parallelism; 6a) has not been implemented herein.

B. Two-Level Parallelism

Figure 6 shows an example of two-level parallelization. Only one case is presented. Other cases involving parallel FEA are being pursued. For instance, Fig. 6a shows how parallel FE and GE managers can be nested. In this paper, we restrict ourselves to parallel DF and LS as shown in Fig. 6b. In this case, multiple processors are used to simultaneously perform parallel LS along all available directions. Thus it is necessary to specify the number of processors to be used for parallel LS along a single direction. The minimum number of processors available must equal the number of processors to be used for parallel LS per direction. This gives rise to a two-level implementation. Consider the scenario shown in Fig. 6b. Processors P1–P4 perform FEs for their assigned step size along their assigned direction vector. LS managers P1 and P3 collect step size, objective function, and maximum constraint values and use these to determine the optimum step size along the assigned direction. DF manager P1 collects all the optimum step sizes and determines the best step size based on the best objective function value. The next section discusses the details of the numerical examples that have been solved using the preceding implementation.

IV. Numerical Examples

Solutions to two example problems that are solved using the developed software are presented in this section. All computations were carried out using the Arizona State University (ASU) FEM cluster. The ASU cluster is an eight-node cluster running Microsoft Windows 2000 and MPI. The nodes are connected by a Cisco Catalyst 3550-12T gigabit switch. A typical node is dual-processor Intel Pentium 4-1.7 GHz Xeon with 1 GB RAM, an Intel gigabit ethernet card and an IDE hard disk. TCP/IP communication protocol is used. All timings are wall clock timings. The term 1PN denotes execution when one process is launched per node in a cluster. Similarly, the term 2PN denotes execution when two processes are launched per node in a cluster. In addition, the following

nomenclature (command line switches) is used to indicate the execution mode:

1) −ge, parallel GE (gradient evaluation) is carried out using all available processors.

2) −df: d, parallel DF is carried out using d processors. Thus, d direction vectors are used (note: line search is simultaneously conducted along each direction).

3) −ls: ℓ, parallel LS is carried out using ℓ processors along each direction if ":d" is specified, otherwise, sequentially one direction after another. Note that the condition PE $\geq \ell$ is necessary for carrying out parallel line search.

A. Sizing Optimization of a Planar Truss (ROD-T1581)

The FE model is shown in Fig. 7. Horizontal loads are applied to all the nodes on the left side of the truss whereas the bottom nodes of the truss are completely constrained. The model has 1581 nodes and 4550 truss elements. The optimization problem is to minimize the volume of material used with axial stress constraints in each of the truss members. There are 150 sizing design variables (cross-sectional areas). The time for one function evaluation using a single processor is less than 1 s. For this model, there is no advantage in carrying out the FE analysis in parallel. The design optimization results are presented in Tables 1 and 2. Design optimization runs were made for 20 iterations.

B. Discussion of the Results

The beneficial effect of using multiple search directions is evident from numerical experience on this problem. After 20 iterations, $f = 53,455$ in^3 compared to the 18,000 value when using four directions. Detailed conclusions are presented next.

1. *Parallel Gradient Evaluation (*−ge*)*

There is a consistent reduction in parallel GE time with increasing number of processors. For parallel GE with eight processors, the number of design variables handled by each processor is $[(150/8) + 1] = 19$. Thus we have an expected speedup of $(150/19) = 7.9$. The speedup obtained is $(990/173) = 5.72$. This difference between expected and obtained speedups can be attributed to increased communication time because of the large number of active constraints, which result in large constraint gradient vectors that have to be communicated and assembled.

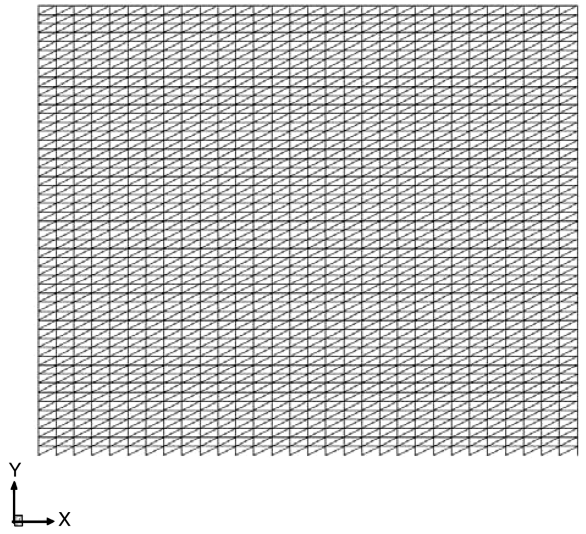


Fig. 7 Finite element model of the planar truss.

2. *Parallel Direction Finding (*−df*: 4)*

Because parallel DF is done in two stages, processors P1 and P2 calculate the standard and tangent directions, respectively, and then the remaining directions (γ-directions) are calculated simultaneously in the second stage. Thus the process of generating four directions in parallel is reduced to a two-stage process leading to an expected speedup of $(4/2) = 2.0$, which is consistent with the observed speedup of $(1442/754) = 1.91$. The number of active constraints in this problem is very high and thus DF time dominates. Thus, the overall speedup follows the DF trend yielding a speedup of $(2690/1025) = 2.62$.

3. *Parallel Line Search (*−ls*)*

When only parallel line search is specified, all available processors (in this case, eight) are used to perform parallel LS along a each of the four directions, one at a time. For this problem, speedup with eight processors is $(250/98) = 2.55$.

4. *Combined Parallel Direction Finding and Line Search (*−df*: 4–*ls*: 2)*

Parallel LS specified along with parallel DF provides a total speedup of $(250/48) = 5.21$. Here, two processors are used along each of the four directions, thus using a total of eight processors. When −df: 4–ls: 4 , we have a gain of 8.3 compared with a single

Table 1 Design optimization results from ROD-T1581 problem (20 design iterations)[a,b]

No. of processors	Parallel computations	Time, s				Normalized timing	Final objective function, in^3
		GE	DF	LS	Total		
1	N/A (one search direction)	999	5	64	1074	N/A	53,455
1	N/A	990	1442	250	2690	2.62	18,629
8 (1PN)	−ge	173	1453	259	1890	1.84	18,629
8 (1PN)	−ge–ls	170	919	98	1196	1.17	18,301
8 (1PN)	−ge–df: 4	175	754	97	1031	1.01	18,629
8 (1PN)	−ge–df: 4 − ls: 2	177	796	48	1025	1.00	18,186
16 (2PN)	−ge–df: 4 − ls: 4	97	983	30	1117	1.09	18,422

[a]Four search directions used in each case (except as noted in first row) [b]Convergence not obtained with single direction as discussed in Section II

Table 2 Function evaluation results for $\mathrm{ROD_{T1581}}$ problem (20 design iterations)

No. of processors	Parallel computations	$\mathrm{NFV_{GE}}$	$\mathrm{NFV_{LS}}$	$\mathrm{NFV_{LS}/PE}$	$\mathrm{NFV_{Total}}$
1	NA	3000	755	N/A	3798
8 (1PN)	−ge	3000	755	N/A	3798
8 (1PN)	−ge–ls	3000	1824	228(1824/8)	4847
8 (1PN)	−ge–df: 4	3000	728	182 (728/4)	3751
8 (1PN)	−ge–df: 4 − ls: 2	3000	792	99 (792/8)	3815
16 (2PN)	−ge–df: 4 − ls: 4	3000	1040	65 (1040/16)	4063

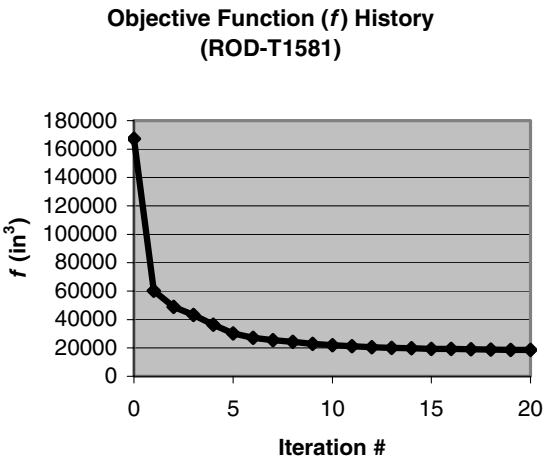


Fig. 8 Design history for planar truss.

processor. However, speedup per direction equals $(8.3/4) = 2.1$ going from 1 to 16 processors. This shows that future research is needed in faster parallel algorithms for line search.

5. Overall Speedup

Figure 8 shows the design history. Very little change in the objective function is observed after the 12th iteration. The best overall speedup of 2.62 is obtained with eight processors (1PN). Using 16 processors does not yield any improvement over the eight-processor case; the gains from parallel GE are lost due to the increase in parallel DF time, probably due to memory bus bottleneck that shows up when both the processors try to access memory at the same time. Note that in this problem, DF (i.e., LP solution) dominates in time.

To gain a better understanding of the computational issues, Table 2 shows the number of function evaluations. It should be noted that NFV_{LS} includes the function evaluations by all processors even if the FEs occur simultaneously. Thus an additional column of function evaluations per processor (NFV_{LS}/PE) is provided to give an idea of the reduction in NFV_{LS} due to parallelization. If one assumes that FEs take all the compute time, then the expected speedup should be $[3798/(375 + 99)] = 8.01$. However, the actual speedup is 2.62. This is because the problem is small with respect to solution of Eq. (2).

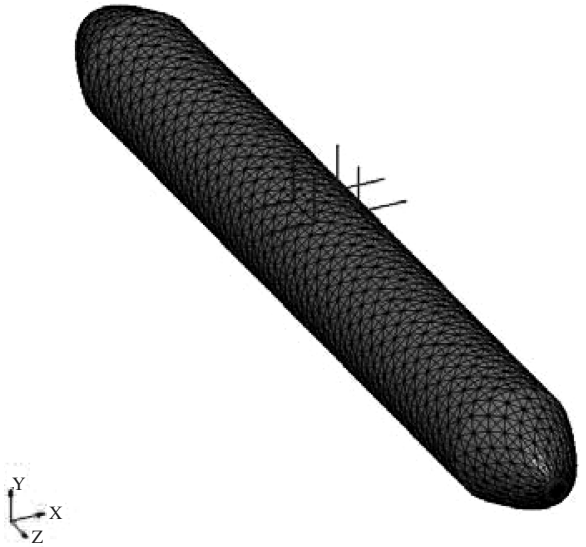


Fig. 9 FE model showing elements and boundary conditions.

C. Sizing Optimization of a Pressure Vessel (TB3- T2839-40)

The FE model with the boundary conditions is shown in Fig. 9. The tank is loaded with a uniform pressure. The model has 2722 nodes and 5440 three-noded thin shell elements. The optimization problem is to minimize the volume of material used with von Mises stress constraints. There are 40 sizing design variables (shell thicknesses). The time for one end-to-end FE analysis using a single processor is 12 s. For this model, there is no advantage in carrying out the FE analysis in parallel. The design optimization results are presented in Tables 3 and 4. Design optimization runs were made for 30 iterations.

D. Discussion of the Results

1. Parallel Gradient Evaluation (−ge)

As before, there is a reduction in parallel GE time with increasing number of processors. For only parallel GE with eight processors, the number of design variables handled by each processor is $(40/8) = 5$. Thus we have an expected speedup of $(40/5) = 8$. The speedup obtained is $(4826/761) = 6.34$. This difference between expected and obtained speedups can be attributed to increased communication time because of the large number of active constraints, which result in large constraint gradient vectors that have to be communicated and assembled.

Table 3 Design optimization results from TB3-T2839-40 problem (30 design iterations)

No. of processors	Parallel computations	Time, s				Normalized timing	Final objective function, m^3
		GE	DF	LS	Total		
1	N/A	4826	409	3538	8896	7.10	0.581
8 (1PN)	−ge	761	408	3503	4792	3.82	0.581
8 (1PN)	−ge–ls	748	422	974	2265	1.81	0.593
8 (1PN)	−ge–df: 4	750	205	1091	2166	1.73	0.581
8 (1PN)	−ge–df: 4 − ls: 2	758	205	1091	2166	1.73	0.581
16 (2PN)	−ge–df: 4 − ls: 4	534	189	409	1254	1.00	0.591

Table 4 Function evaluation results for TB3-T2839-40 problem (30 design iterations)

No. of processors	Parallel computations	NFV_{GE}	NFV_{LS}	NFV_{LS}/PE	NFV_{Total}
1	N/A	1200	876	N/A	2076
8 (1PN)	−ge	1200	876	N/A	2076
8 (1PN)	−ge–ls	1200	1880	235(1880/8)	3113
8 (1PN)	−ge–df: 4	1200	896	224 (896/4)	2129
8 (1PN)	−ge–df: 4 − ls: 2	1200	984	123 (984/8)	2217
16 (2PN)	−ge–df: 4 − ls: 4	1200	1440	180 (1440/16)	2673

*2. Parallel Direction Finding (−*df*: 4)*

Parallel DF is done in two stages, same as for the previous problem. Again, the expected speedup is $(4/2) = 2.0$, which is consistent with the observed speedup of $(409/205) = 2.0$.

*3. Parallel Line Search (−*ls*)*

For parallel line search, using eight processors along each of the four directions one at a time, the obtained speedup is $(3538/974) = 3.63$.

*4. Combined Parallel Direction Finding and Line Search (−*df*: 4–*ls*: 2)*

Total speedup of parallel LS along with parallel DF is $(3538/552) = 6.41$. In this case, eight processors are available for LS along four directions. Thus two processors are specified per line search so that all available processors can be used. This gives speedup per direction as $(6.41/4) = 1.6$. With −df: 4–ls: 2 , we have speedup of 8.65 for four directions, or 2.16 per direction.

5. Overall Speedup

Figure 10 shows the design history. Very little change in the objective function is observed after the 10th iteration. Unlike the previous problem, the DF time does not dominate the solution. Because GE and LS require most of the computation time and parallel GE and combined parallel DF and LS provide significant speedup, the overall time is also greatly reduced with a speedup of $(8896/1594) = 5.58$. When both the processors are used per node, the speedup is further increased to 7.10. Once again, unlike the previous problem, the gains from parallel GE and parallel LS increase the efficiency of the solution. The DF problem is considerably smaller because the number of active constraints is smaller. If one assumes that FEs take all the compute time, then the expected speedup should be $[2076/(150 + 123)] = 7.6$. However, the actual speedup is 5.58, a considerably improved performance compared with the previous problem.

V. Conclusions

An integrated design optimization system has been developed for both sequential and parallel processing using message-passing interface. The results are promising and show that the methodology can be used on relatively small clusters to achieve performance gains. In the future, when considering larger FE models, parallel FEA can be integrated into the parallel optimization in this paper with further gains; in fact, the use of parallel FEA will result in multiplicative speedups because the parallelization is nested (three-level). The parallelization schemes used here are generalizable to other constrained gradient methods. However, it should be noted that obtaining significant speedups require attention to both algorithms and computer implementation, as the latter is especially important in parallel computations. Finally, it is important to clarify what is claimed and not claimed in this paper.

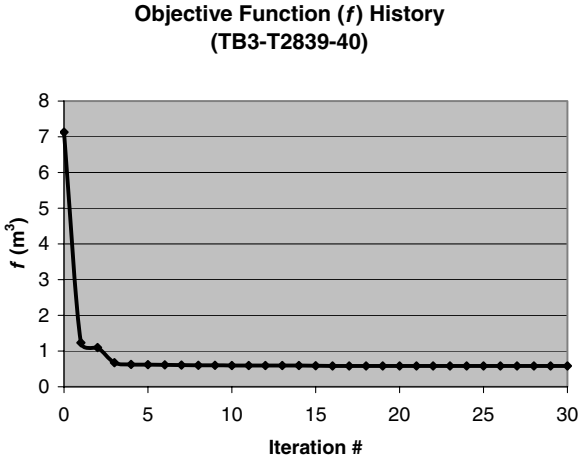


Fig. 10 Design history for pressure vessel problem.

Claimed:

1) In absolute terms, significant gains in total CPU time are obtainable through parallelization of various steps in the gradient-based optimization procedure. For instance, a gain of about 7:1 has been obtained using 16 processors on one test problem in this paper.

2) Reduced CPU time is not the only benefit in parallel optimization as presented. The idea of generating a family of search directions in *any gradient algorithm*, as per Eq. (5), results in a robust exploration of the design space for a better optimum (in the algorithm here, helps in nonoptimal solutions resulting from zig-zagging or jamming on several test problems including those not included here). This is relevant because proper scaling of the problem is often not possible.

3) *Multilevel* parallelization as implemented here demonstrates promise for solution of large-scale engineering optimization, through careful attention to individual computational steps.

Not Claimed:

1) Near linear speedups are possible in every step. Gradient evaluation by forward differences yields linear speedup, but line search using the Avriel routine is sublinear. Thus, optimum allocation of processors among different tasks is important. For instance, if the finite element model is truly large, it may be best to parallelize only the function evaluations.

2) The specific algorithms used here are best for parallel implementation. Nongradient and/or stochastic techniques for may outperform those considered here, in massively parallel computers, an area that needs research.

Acknowledgment

The first author wishes to thank the Chandler, AZ division of Intel Corporation for their kindness in donating most of the machines in the Arizona State University FEM cluster.

References

[1] Panel Session, "Large Scale and High Performance Optimization," Session 25, *10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Albany, NY, Sept. 2004.

[2] Dongarra, J., Sterling, T., Simon, H., and Strohmaier, E., "High Performance Computing: Clusters, Constellations, MPPs, and Future Directions," *IEEE Computing in Science & Engineering*, Vol. 7, No. 2, 2005.

[3] Benson, S., McInnes, L., More, J., and Sarich, J., "Scalable Algorithms in Optimization: Computational Experiments," AIAA Paper 2004-4450, 2004.

[4] Burgee, S., Giunta, A. A., Narducci, R., Watson, L. T., Grossman, B., and Haftka, R. T., "A Coarse Grained Parallel Variable-Complexity Multidisciplinary Optimization Paradigm," *International Journal of Supercomputer Applications and High Performance Computing*, Vol. 10, No. 4, 1996, pp. 269–299.

[5] Grauer, M., and Barth, T., "Grid Based Computing for Multidisciplinary Analysis and Optimization," AIAA Paper 2004-4456, 2004.

[6] Leite, J. P. B., "Parallel Simulated Annealing for Structural Optimization," *Computers and Structures*, Vol. 73, Nos. 1–5, 1999, pp. 545–564.

[7] Padula, S. L., and Stone, S. C., "Parallel Implementation of Large-Scale Structural Optimization," *Structural Optimization*, Vol. 16, Nos. 2–3, 1998, pp. 176–185.

[8] Venter, G., and Watson, B., "Efficient Optimization Algorithms for Parallel Applications," AIAA Paper 2000-4819, 2000.

[9] Rajan, S. D., and Nguyen, D. T., "Design Optimization of Discrete Structural Systems Using MPI-Enabled Genetic Algorithm," *Journal of Structural and Multidisciplinary Optimization*, Vol. 28, No. 5, 2004, pp. 340–348.

[10] APPSPACK, Sandia National Labs, http://software.sandia.gov/appspack/index.html.

[11] Leite, J. P. B., "Parallel Simulated Annealing for Structural Optimization," *Computers and Structures*, Vol. 73, Nos. 1–5, 1999, pp. 545–564.

[12] Byrd, R. H., Schnabel, R. B., and Schultz, G. A., "Parallel Quasi-Newton Methods for Unconstrained Optimization," *Mathematical*

Programming, Vol. 42, 1988, pp. 273–306.
[13] High, K. A., and LaRoche, R. D., "Parallel Nonlinear Optimization Techniques for Chemical Process Design Problems," *Computers & Chemical Engineering*, Vol. 19, Nos. 6–7, 1995, pp. 807–825.
[14] Gorka, A., and Kostreva, M. M., "Probabilistic Version of the Method of Feasible Directions," *Applied Mathematics and Computation*, Vol. 130, 2002, pp. 253–264.
[15] Grauer, M., and Pressmar, D. B. (eds.), "Parallel Computing and Mathematical Optimization," *Proceedings of the Workshop on Parallel Algorithms and Transputers for Optimization*, Lecture Notes in Economics and Mathematical Systems, Vol. 367, Springer–Verlag, Berlin, 1991.
[16] Belegundu, A. D., Damle, A., Rajan, S. D., Dattaguru, B., and St. Ville, J., "Parallel Line Search in Method of Feasible Directions," *Optimization and Engineering*, Vol. 5, No. 3, 2004, pp. 379–388.
[17] Rajan, S. D., Belegundu, A. D., Lee, D., Damle, A., and St. Ville, J. A., "Finite Element Analysis and Design Optimization in a Distributed Computing Environment," AIAA Paper 2004-4453, 2004.
[18] FOMASS, Optix Software System, http://www.fomaas.de/intro.htm.
[19] DAKOTA: A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis, Sandia National Labs, http://endo.sandia.gov/DAKOTA/ software.html.
[20] TAO: Toolkit for Advanced Optimization, Argonne National Labs, http://www-unix.mcs.anl.gov/tao/.
[21] LANCELOT: A Package for Large-Scale Nonlinear Optimization, http://www.numerical.rl.ac.uk/lancelot/ blurb.html.
[22] HYI-3D: A Software System for Volumetrically Controlled Manufacturing, HYI, Inc., Phoenix, AZ, 2005, http://enpub.fulton.asu.edu/structures/rajan/32vs64perf.htm.
[23] Zoutendijk, G., *Method of Feasible Directions*, Elsevier, New York, 1960.

E. Livne
Associate Editor