A Parallel Optimization Primer

Michael Lewis
Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia

Virginia Torczon
College of William & Mary
Williamsburg, Virginia

Center for Research on Parallel Computation

Virginia Torczon: `va@cs.wm.edu`

Michael Lewis: `buckaroo@icase.edu`

These slides, links to other optimization sites, and other materials:

`http://www.icase.edu/~buckaroo/ihop/guide.html.`

Electronic mailing list (please sign up).

The general computational optimization problem

Given a set $\Omega \subset I\!R^n$ and a function $f(x)$, find $x \in \Omega$ that minimizes $f(x)$.

Convention: We will discuss minimization. Maximizing $f(x)$ is equivalent to minimizing $-f(x)$.

This course: How can we apply computational parallelism to solving this problem in some of its more common manifestations?

The state of parallel optimization

Good news:

- The "obvious" things are the most effective.

- For many problems, linear algebra is not the bottleneck.

- Parallelism in optimization can peacefully coexist with parallelism in user code.

The state of parallel optimization

Bad news:

- Many of the "obvious" things have definite limits to scalability.
- The simple things that scale might not provide any benefit.
- There is little parallel optimization software publicly available.

This course

We will focus on the simple things you can do to parallelize optimization, because:

- they are the most effective and
- you will probably have to do them yourself.

We will also discuss the question of scalability in optimization algorithms.

This course

We will *not* focus on linear algebra. Why not?

- **Generality:** The effectiveness of parallel linear algebra depends on problem structure and the parallel and sequential architecture.

- **Granularity:** For many optimization problems, the concurrency of the linear algebra is not sufficient to justify parallelism.

For linear and convex programming linear algebra is a significant cost (though not the only one).

For many nonlinear optimization problems the evaluation of the objective and constraints is the expensive portion of the computation, and the portion most amenable to parallel computation.

Outline

1. Types of optimization problems

2. Linear optimization

   - The simplex method
   - Interior point methods

3. Nonlinear optimization

   - Pseudo-random search algorithms
   - Pattern search algorithms
   - Quasi-Newton algorithms

4. Global optimization

5. Decomposition ideas

6. Software

Preliminaries

The general computational optimization problem

Given a set $\Omega \subset I\!R^n$ and a function $f(x)$, find $x \in \Omega$ that minimizes $f(x)$.

Convention: We will discuss minimization. Maximizing $f(x)$ is equivalent to minimizing $-f(x)$.

We will refer to x as the optimization or design or control or decision variables; $f(x)$ is the *objective*.

Ω is the *feasible region*. If $x \in \Omega$ it is *feasible*; otherwise it is *infeasible*.

Typically Ω is defined by a set of equalities and inequalities.

A solution x_* is a *minimizer*; $f(x_*)$ is the *minimum*.

Specific types of objectives and constraints

Linear:

$$a^T x = \sum_{i=1}^{n} a_i x_i$$

Quadratic:

$$\frac{1}{2} x^T Q x + b^T x$$

where $Q \in I\!R^{n \times n}$ is symmetric.

Convex:

$$\phi(tx + (1-t)y) \leq t\phi(x) + (1-t)\phi(y)$$

Vector inequalities are defined componentwise, e.g., $x \geq 0$ means $x_i \geq 0$, $i = 1, \ldots, n$.

Simple bounds:

$$\ell_i \leq x_i \leq u_i$$

Linear optimization problems

Linear programming (LP): linear objective and linear constraints (in standard form)

$$
\begin{array}{ll}
\text{minimize} & c^T x \\
\text{subject to} & Ax = b \\
 & x \geq 0
\end{array}
$$

Integer programming (IP): LP + the constraint that the variables are only integer-valued.

Mixed integer programming (MIP): LP + both integer and continuous variables.

Nonlinear optimization problems

Convex programming (CP): Convex objective and convex constraints (convex feasible region)

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & g(x) \leq 0 \end{array} \quad .$$

General quadratic programming (QP): quadratic objective and linear constraints

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}x^T Q x + c^T x \\ \text{subject to} & Ax \leq b \end{array} \quad .$$

QPs may or may not be convex.

Nonlinear programming (NLP): general nonlinear objective and constraints

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & g(x) \geq 0 \\ & h(x) = 0 \end{array} \quad .$$

Digression: Complexity

Linear programming:	polynomial time
Convex programming:	polynomial time
Integer programming:	NP-complete
Non-convex QP:	NP-complete
Nonlinear programming:	???

Non-convex QPs routinely appear as subproblems in NLP.

What is “large-scale” optimization?

One view: “large-scale” means a large number of optimization variables.

This course: “large-scale” means a large computational cost.

Small problems can be expensive if there is a large number of implicit state variables $u = u(x)$ which must be computed. For instance, u might describe the physical behavior of the problem.

Example: Flow over a complex airfoil with Mach number and angle of attack as the design variables.

There are only two optimization variables, but many hours per optimization iteration are needed for the solution of the flow equations for u.

Large, sparse problems

- Linear and integer programming.
- Quadratic programming (typically convex).
- Convex programming / semidefinite programming.
- Variational principles associated with PDE.
- Simultaneous analysis and design.

Small/large dense problems

- Problems governed by differential and integral equations, etc.
- Eigenvalue optimization.
- Systems governed by physical simulations.
- Systems governed by engineering analyses.
- Parameter estimation and inverse problems.

Bloated optimization problems

Some problems involving large numbers of optimization variables are “bloated” rather than “large-scale”.

Prototype: underdetermined least-squares,

$$\text{minimize} \quad \tfrac{1}{2} \| Ax - b \|^2$$

$$\text{minimize} \quad \tfrac{1}{2} \| F(x) - b \|^2$$

where A or the Jacobian of F is effectively rank-deficient but there are a large number of variables x.

Such problems arise in parameter estimation and inverse problems where the unknown parameters are parameterized too finely for the problem.

The over-discretization does not add any real information or resolution, can degrade the solution, and increases the computational cost.

Beware this possibility.

The generic optimization algorithm

Given an initial iterate x_0, for $k = 0, 1, \ldots$ until convergence (or termination) do $\{$

1. Find a set X_k of potential next iterates.

2. If $x_+ \in X_k$ is acceptable, $x_{k+1} = x_+$; otherwise, $x_{k+1} = x_k$.

3. Update any necessary information.

$\}$

The possibility that the iteration may fail to produce improvement is a distinctive feature of optimization problems.

Information we can use to assist in the search:

1. 0th-order methods: values of the objective and constraints

2. 1st-order methods: first derivatives (linearization) of the objective and/or constraints

3. 2nd-order methods: second derivatives (curvature) of the objective and/or constraints

4. higher-order methods: tensor methods

(Is happiness one more term in the Taylor's series?)

Sources of computational cost

Linear optimization: the number of variables and constraints

1. Impact on linear algebra.

2. Impact on number of iterations required.

Sources of computational cost

Nonlinear optimization:

1. The nonlinear objective and constraints, and/or their derivatives.

2. The number of evaluations required per iteration.

3. The number of iterations required.

4. The number of variables and constraints.

Items 1–3 mean that nonlinear problems with even small numbers of optimization variables may be very expensive to solve.

Opportunities for computational parallelism in optimization

1. Parallelize the individual evaluations of the objective, constraints, and their derivatives.

2. Parallelize the linear algebra in each iteration.

3. Parallelize the sequential algorithm: concurrent evaluation of the objective and constraint that might be needed (preemptive computation).

4. Use the availability of additional processors to do computations not done in the sequential algorithm in the hopes of solving the problem more quickly (supererogatory computation).

5. Problem decomposition.

The term "speculative function evaluation" is sometimes used to refer jointly to preemptive and supererogatory evaluation.

Opportunities for parallelism: linear optimization

- Sparse linear algebra.
- Branch and bound for integer programming.
- Decomposition.

Opportunities for parallelism: nonlinear optimization

- Concurrent evaluation of objective and constraints and their derivatives.

- Linear algebra.

- Preemptive evaluation of objective and constraints:

  Introducing more concurrency by anticipating all possible paths through the logic of the sequential algorithm.

- Supererogatory evaluation of objective and constraints:

  Computing more function and constraint values than a sequential implementation in hopes of accelerating the search.

- Decomposition.

Linear optimization

Linear programming

Primal problem in standard form:

$$
\begin{array}{ll}
\text{minimize} & c^T x \\
\text{subject to} & Ax = b \\
& x \geq 0
\end{array}
$$

$A \in I\!R^{m \times n}$, $m < n$.

Dual problem:

$$
\begin{array}{ll}
\text{maximize} & b^T y \\
\text{subject to} & A^T y + z = c \\
& z \geq 0
\end{array}
$$

Parallelization of the linear algebra has provided the main benefit to date in linear programming.

Operating assumptions: linear and convex programming

- the objective and constraints are "cheap" to evaluate (sparse dot products and sparse matrix-vector products),
- the number of variables n and (possibly) constraints m is large,
- the matrices involved are typically sparse, and
- a great many iterations k may be required.

The implication is that the expensive portion of the calculation at each iteration lies with the linear algebra calculations:

- sparse matrix-vector and matrix-matrix products,
- factoring or updating the factorization of a large, sparse matrix, and
- using the factorization to solve linear systems of equations.

Note: fine granularity; uncertain degree of concurrency (the facts of life in sparse linear algebra).

The simplex method for linear programming

The solution of an LP is at a vertex of the polyhedron that is the feasible region. The simplex method intelligently travels from vertex to vertex using a form of steepest descent.

One iteration of the revised simplex method: Given a basis B (an $m \times m$ invertible sub-block of A),

1. Solve an $m \times m$ system $B^T \pi = c_B$ (where c_B corresponds to the columns associated with B) for the shadow costs π.

2. Compute the reduced cost for all non-basic variables: $d_j = c_j - \pi^T a_j$ (up to $n - m$ inner products of sparse n-vectors).

3. Choose an entering column based on the reduced cost ("pricing rule") and check optimality.

4. Solve an $m \times m$ system $B\tilde{a}_j = a_j$ to update the entering column.

5. Determine an exiting column of B by the ratio test (m floating point divides).

The fact that only one variable changes in B means that we can perform a rank-1 update rather than re-factor B at each iteration. Moreover, B might not be so large if $m \ll \infty$.

This limits the effectiveness of parallelizing the factorization.

One can use parallelism to apply pricing rules that are too expensive for sequential processing (e.g. greatest decrease), but the benefits are not clear.

The simplex method for linear programming

General conclusion: the simplex method does not parallelize well. One may be better off using a super-scalar workstation.

Interior point methods for linear programming

Interior point methods move towards the solution along a path that is not on the boundary of the feasible region.

One iteration of the Mehrotra primal-dual predictor-corrector method: Given x from the primal LP and y, z from the dual LP,

1. Form the $m \times m$ matrix $A\Theta A^T$, where Θ is diagonal.

2. Compute the LDL^T (Cholesky) decomposition of $A\Theta A^T$.

3. Compute the predictor directions $(\Delta\hat{x}, \Delta\hat{y}, \Delta\hat{z})$; this requires solving a linear system involving $A\Theta A^T$ and a matrix-vector product with A^T (m inner products of sparse n-vectors).

4. Do a ratio test to compute $\hat{\alpha}_P$ and $\hat{\alpha}_D$ (n floating point divides).

5. Compute $\mu = \mu(x, y, z, \Delta_P, \hat{\alpha}_P, \hat{\alpha}_D)$.

6. Compute the search direction $(\Delta x, \Delta y, \Delta z)$; this requires another back-solve involving $A\Theta A^T$ and a matrix-vector product with A^T (n inner products of sparse m-vectors).

7. Do another ratio test to compute α_P and α_D (n floating point divides).

8. Update: $x_{k+1} = x_k + \alpha_P \Delta x$, $y_{k+1} = y_k + \alpha_D \Delta y$, $z_{k+1} = z_k + \alpha_D \Delta z$.

Parallelism in interior point methods

Parallel sparse Cholesky algorithms can be brought to bear on $A\Theta A^T$.

A parallel version of the Mehrotra primal-dual predictor-corrector interior point method in the CPLEX package (Bixby, Lustig, et al.) is available for the SGI Challenge (using sparse Cholesky algorithms by Rothberg and Hendrickson).

Lustig and Rothberg (1996) reported speed-ups of 5–10 on 16 processors.

Coleman, et al. (1997) reported speed-ups of 3–4 on 16 processors of an SP2 for a parallel version pPCx of the PCx interior point code. They parallelized only the Cholesky factorization.

Parallelization of the CPLEX interior point code

Sequential steps:

1. LP preprocessing.

2. Row ordering of A to prevent fill.

3. Symbolic factorization of AA^T.

4. Computing an initial iterate.

Parallelization of the CPLEX interior point code

Parallel steps:

1. Forming $A\Theta A^T$.

2. Factoring $A\Theta A^T$.

3. Solving for Δy and $\Delta\hat{y}$ using the LDL^T factorization.

4. The ratio tests.

5. Various matrix-vector products involving A and A^T.

Not necessarily "massively parallel," but effective.

Integer programming

The general pure integer program in standard form:

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax = b \\ & x \geq 0 \\ & x \text{ integral} \end{array}$$

where A is a matrix of order $m \times n$.

Operating assumptions: integer programming

- the objective and constraints are cheap to evaluate,
- the number of variables n and (possibly) constraints m is large,
- the matrices involved are typically sparse, and
- a *great many* iterations k may be required ($\gg$ than for the corresponding LP).

The implication is that the expensive portion of the calculation at each iteration lies with the solution of the subproblems:

- finding the optimal solution of an LP relaxation, which is itself large and sparse,
- possibly transferring entire LPs between processes to enable "hot starts" for the new subproblems, and,
- if cutting plane techniques are employed, generating cuts and managing these cuts using a cut pool.

Note: coarse granularity; modest concurrency; potential for significant data movement.

The branch and bound method for integer programming

The branch and bound method uses the fact that an optimal solution (if one exists) can be found at a vertex of the IP polyhedron (the convex hull of integer points in the feasible region $Ax = b$, $x \geq 0$).

Branch and bound is a special divide and conquer technique.

The IP is relaxed by dropping the integrality constraints to form the so-called LP relaxation.

The LP relaxation is solved using a version of the simplex method to produce a vertex solution and a bound on the best possible solution for the IP.

Branch and bound then builds a search tree as follows:

1. If the LP relaxation is infeasible, return. The node is fathomed; any subproblems will also be infeasible.

2. If the solution is integral, return. The node is fathomed; the solution is a feasible candidate for the optimal solution and may be used to improve the bound on the best possible solution for the problem.

3. If the solution bound on the optimal value for the LP relaxation is not better than the current bound (the best feasible solution discovered thus far) return. The subproblem can be "pruned" from the search tree because additional constraints will not decrease the value of the objective.

4. Else choose a variable x_j in the current solution x that is not an integer. Let $\ell_j = \lfloor x_j \rfloor$ and $u_j = \lceil x_j \rceil$.

   (a) Create two new LP relaxations of the original problem:

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax = b \\ & x \geq 0 \\ & x_j \leq \ell_j \end{array}$$

   and

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax = b \\ & x \geq 0 \\ & x_j \geq u_j. \end{array}$$

   Restore both LP relaxations to standard form.

   (b) Repeat the process with the two new LP relaxations.

Parallelism in integer programming

The divide-and-conquer nature of branch-and-bound suggests obvious parallelism: each of the LP subproblems created by a branch can be solved independently.

Parallelism in integer programming

Implementation must address:

- Dynamic construction of the search tree processes.

- Load balancing: each LP relaxation is solved using an iterative process (e.g., a version of the simplex algorithm) that completes with a varying number of iterations that depends on the particular subproblem.

  Furthermore, it is impossible to predict in advance when subproblems will be either pruned or fathomed so that future branching at that node is no longer necessary.

- Potential cost of data transfers associated with "hot starts" for the subproblems.

Not necessarily "massively parallel," but effective.

Branch and cut methods

Applicable to IPs with special structure, such as the Traveling Salesman Problem (TSP), Vehicle Routing, and Set Partitioning (crew scheduling).

These approaches incorporate cutting plane methods, which begin with a relaxation of the IP and generate and add linear constraints (cuts) violated by the optimal solution of the current relaxation, but not violated by any integer solution. The goal is to better define ("narrow down") the IP polyhedron (the convex hull of integral solutions) in the neighborhood of its optimal solutions.

Incorporate strategy within branch-and-bound, but now allow branching based on the cuts (i.e., the constraints) rather than simply on the variables.

Parallel branch and cut methods

Use independent processes to generate the cuts and maintain the cuts in so-called "cut pools". These processes can run more-or-less concurrently with the solution of the LP relaxations, with the same general performance profile and caveats.

Why bother? Because "the branch-and-cut algorithm, when implemented with model-specific cuts, is the most effective solution procedure presently known for difficult combinatorial optimization models" (such as the Traveling Salesman Problem).

Applegate, Bixby, Chvatal, Cook have successfully applied parallelism to solve large TSPs on both workstation networks and the SGI Challenge.

Their current world record is a 7397-city TSP solved using a cluster of 60 workstations followed by a cluster of 10 20-processor SGIs.

Coming August 1997: fully parallel generic branch-and cut implementation from the Cornell Theory Center by Trotter, et al.

Nonlinear optimization

The nonlinear optimization problem:

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & g(x) \geq 0 \\ & h(x) = 0 \quad . \end{array}$$

General nonlinear objective and constraints.

An important distinction

Global convergence vs. convergence to a global minimizer:

- **Global convergence:** Convergence to a stationary point or local minimizer from an arbitrary starting point.

- **Convergence to a global minimizer:** Convergence to a global minimizer over the feasible region.

In linear and convex programming, one gets global convergence to a global minimizer. In nonconvex nonlinear optimization, this is not necessarily the case.

We have ways to insure global convergence and it is relatively easy to achieve in practice. Convergence to a global minimizer is both difficult to insure and computationally expensive.

The grand division of nonlinear optimization techniques

1. Methods that do not use derivatives ...
   - Pseudo-random search algorithms
   - Pattern search algorithms

2. ... and those that do
   - Quasi-Newton algorithms

Hybrids of these two classes are obviously possible.

Parallelism and the grand division of nonlinear optimization techniques

For the quasi-Newton algorithms, almost all the useful parallelism was foreseen at least as early as 1980. This is because the most useful general parallelism is the obvious parallelism, and quasi-Newton algorithms had already received close attention in the sequential case.

The non-derivative methods—genetic algorithms and pattern search methods—only came into their own in the 1980s and 1990s. They present opportunities for parallelism not present in the quasi-Newton methods.

Operating assumptions: nonlinear (nonconvex) programming

- the objective and/or constraints is/are expensive to evaluate,
- the first derivatives (when required) also are expensive to evaluate,
- typically the number of optimization variables n and constraints m is relatively modest (in comparison to the numbers typically found in LP/IP) or the problem is sparse and not too nonlinear, and
- there may be a large number of optimization iterations k.

The implication is that the expensive portion of the calculation at each iteration lies in the evaluation of the objective, constraints, and gradient values.

We will, however, also consider the case where the linear algebra may be expensive.

Note: coarse granularity; potential for a high degree of concurrency.

Application specific parallelism:

- Intra-function evaluations

- Sensitivity calculations (e.g., an adjoint method)

Obviously, we cannot discuss these in any generality. We will focus on general opportunities for parallelism in nonlinear optimization.

Pseudo-random search algorithms: Genetic algorithms

Given an initial collection of candidate solutions X_0 ("population"),
for $k = 0, 1, \ldots$ until *halted* do {

Find a new set X_{k+1} of potential solutions:

1. Draw from X_k according to a distribution weighted in favor of the "better" members ("reproduction").
2. Encode the elements of X_k as bit strings Y_k.
3. Randomly choose pairs from Y_k and swap randomly chosen substrings ("cross-over").

4. Randomly flip bits in randomly chosen elements of Y_k ("mutation").

5. Map the resulting collection of bit strings Y_{k+1} to X_{k+1}.

}

One can think of this process as generating a sequence $\{x_k\}$ where x_k is the best element of X_k.

A public service announcement

It was Herbert Spencer, not Charles Darwin, who spoke of "survival of the fittest."

Darwin's term was "natural selection."

Genetic algorithms

Advantages:

- Simple to implement.
- Simple to parallelize: distribute the population X_k.
- Applicable to nonlinear problems with discrete variables.
- Can be used in the absence of numerical objective and constraints (e.g., when only rank or ordinal values are applicable).

Genetic algorithms

Putative advantages:

- Seemingly unlimited in scalability: increase the size of the population X_k.

- Explores the design space (i.e., it can sample widely).

Genetic algorithms

Disadvantages:

- Slow and inefficient.
- Can be ineffective as the number of design variables or constraints grows.
- No satisfactory convergence analysis.

Convergence analysis of genetic algorithms

There is no guarantee that a genetic algorithm will converge even to a stationary point, much less a local or global minimizer.

With probability 1, the population in a genetic algorithm will eventually stop changing.

Opportunities for parallelism

Genetic algorithms are parallel and scalable (but we will qualify the latter statement later).

At each iteration we can compute the values of the objective and constraints for the search points (i.e., the members of the population) in parallel.

We can scale the algorithm by increasing the number of points in the search population.

Any sequential component (e.g., selecting the best feasible objective value, generating a new population for the next iterate) is generally negligible compared to the cost of objective and constraint evaluations.

Genetic algorithms: parallelism

Sequential algorithm:

```
for k = 0, 1, ...  {
  for x in X_k {       /* Parallel DO-loop */
    compute f(x) and g(x)
  }
  update population X_k.
}
```

We might expect the speed-up to be close to linear.

However, the loop might not be balanced. For instance, if x is infeasible, we might not compute $f(x)$ or parts of $g(x)$.

For this reason, it might be better not to think of the parallelism as data parallelism but as dynamic task parallelism.

Direct search methods: Pattern search algorithms

Examples of pattern search algorithms:

1. Coordinate search with fixed step-size.

2. Evolutionary optimization and its variants (G. E. P. Box).

3. The pattern search method of Hooke & Jeeves.

4. Multidirectional search (Torczon & Dennis).

5. Positive basis and rank ordered pattern search; pattern search for bound and linearly constrained minimization (Lewis & Torczon).

How pattern search algorithms work

Choose $x_0 \in I\!R^n$, $\Delta_0 > 0$.

For $k = 0, 1, \ldots$, until convergence do $\{$

1. Choose $P_k \in I\!R^{n \times p_k}$, where $p_k > n$.

2. If there exists $s_k \in \Delta_k P_k$ for which $f(x_k + s_k) < f(x_k)$, then

   set $x_{k+1} = x_k + s_k$, and choose $\Delta_{k+1} \geq \Delta_k$.

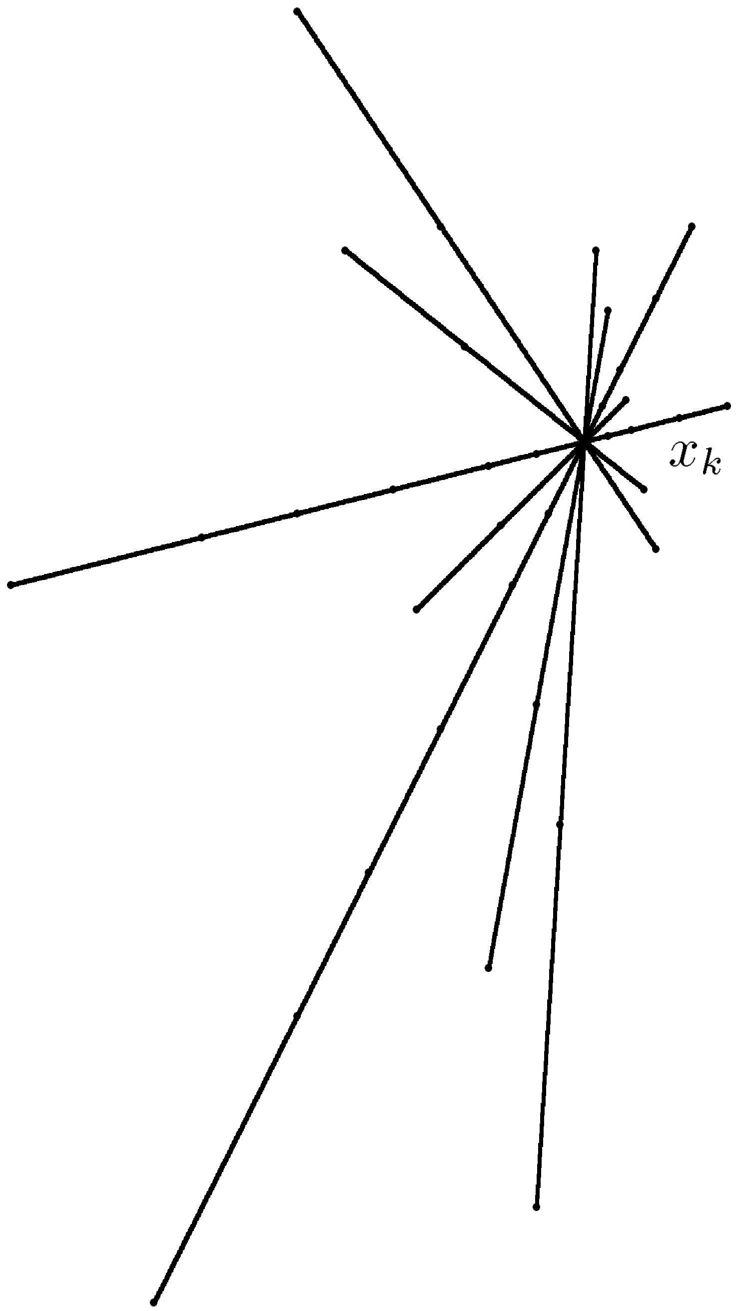
   Else,

   set $x_{k+1} = x_k$ and choose $\Delta_{k+1} < \Delta_k$.

$\}$

The patterns P_k must satisfy certain mild algebraic and geometric conditions, as must the update of Δ_k.

Pattern search in pictures




x_k

Pattern search methods

Advantages:

- Simple to implement.
- Simple to parallelize: distribute the pattern P_k.
- Can be used in the absence of numerical objective and constraints (e.g., when only rank or ordinal values are applicable).

$\star$ Robustness/convergence analysis.

Pattern search methods

Putative advantages:

- Seemingly unlimited in scalability: increase the number of points in the pattern P_k.

- Explores the design space.

Pattern search methods

Disadvantages:

- Can be slow.
- Less efficient as the number of design variables grows.
- Optimal level of scaling not known *a priori*.

Convergence analysis of pattern search algorithms

First-order stationary point convergence: In the unconstrained case, either

$$\liminf_{k \to \infty} \| \nabla f(x_k) \| = 0$$

or

$$\lim_{k \to \infty} \| \nabla f(x_k) \| = 0.$$

In the bound and linearly constrained case, convergence to the KKT (constrained stationary) points.

Opportunities for parallelism

Pattern search algorithms are parallel and scalable.

At each iteration we can compute the values of the objective and constraints for the search points.

We can scale the algorithm by increasing the number of points in the pattern.

Any sequential component (e.g., selecting the best feasible objective value, generating a new pattern) is generally negligible compared to the cost of objective and constraint evaluations.

Pattern search methods: parallelism

Sequential algorithm:

```
for k = 0, 1, ..., until convergence do {
  choose P_k
  for x in part or all of x_k + Δ_k P_k { /* Parallelizable loop */
    compute f(x) and g(x)
  }
  update x_k, P_k and Δ_k
}
```

As we saw for genetic algorithms, the parallel loop might not be balanced. Otherwise, we would expect close to linear speed-up.

A public service announcement: The Nelder-Mead algorithm

The very popular Nelder-Mead algorithm is not quite a pattern search method.

For years, numerical evidence tests indicated that the Nelder-Mead algorithm could stop at points that were not even stationary points (hence, not minimizers), particularly as the number of design variables increased (> 5).

McKinnon has recently given (1996) an explicit example of a convex, differentiable function of two variables for which Nelder-Mead fails.

Moral: Watch out if you use Nelder-Mead.

Parallelism in genetic algorithms and pattern search

- Data parallelism or task parallelism? The parallelism in direct search methods looks like loop-based data parallelism, but is that the best way to think of it?

  Constraints (and iterative solutions and simulations) can lead to load-imbalance, for instance.

- Supererogatory function evaluation does more work (and thus uses more processors), but does it really help?

- Is it better to compute more objective and constraint values at each iteration or fewer values more quickly? (I.e., more information per iteration, or faster iterations?)

Parallelism in genetic algorithms and pattern search

Because both GAs and pattern search methods can incorporate more points into their search strategies ("population", "pattern"), their parallel scalability appears to be less governed by n, the number of optimization variables, than by p, the number of processors available.

However, the improvement in the performance of these algorithms due to extra function evaluations is partly due to the limited efficiency of non-gradient methods.

Moreover, the effectiveness of these algorithms diminishes as n increases (less so for pattern search than for GAs). Thus, increasing p might not offset increasing n.

Genetic algorithms and pattern search, and scalability

GAs appear to have little limit to their scalability, but only because they are inefficient algorithms to begin with.

Pattern search methods benefit from increasing the size of the pattern up to some (unknown) point, after which there is little benefit.

The following slides illustrate these observations.

We compare *untuned* versions of the PGAPACK parallel GA package (Levine) and the PDS parallel pattern search algorithm (Torczon).

DeJong test function F1 (population/iteration = 100)

$$f_1(x) = \sum_{i=1}^{3} x_i^2, \qquad -5.12 \leq x_i \leq 5.11$$

PGAPack			PDS			optimal	
10 bits for the gray-coding			tol = 1.0D-03; x_0 = (5.11,5.11,5.11)				
best x	best f	# iter.	best x	best f	# iter.	x_*	f_*
1.00D-02	6.50D-03	100	-3.52D-04	1.61D-07	16	0	0
-8.00D-02			1.36D-04			0	
-4.00D-02			1.36D-04			0	

DeJong test function F1 (population/iteration = 100)

$$f_1(x) = \sum_{i=1}^{3} x_i^2, \qquad -5.12 \leq x_i \leq 5.11$$

PGAPack			PDS			optimal	
31 bits for the gray-coding			tol = 1.0D-08; x_0 = (5.11,5.11,5.11)				
best x	best f	# iter.	best x	best f	# iter.	x_*	f_*
0.30D+00	8.84D-02	100	5.96D-10	1.06D-18	31	0	0
-2.58D-03			5.96D-10			0	
1.89D-02			5.96D-10			0	

DeJong test function F2 (population/iteration = 100)

$$f_2(x) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2, \qquad -2.048 \le x_i \le 2.047$$

PGAPack			PDS			optimal	
12 bits for the gray-coding			tol = 1.0D-03; x_0 = (-1.2,1)				
best x	best f	# iter.	best x	best f	# iter.	x_*	f_*
0.89D+00	1.14D-02	100	0.98D+00	2.50D-04	74	1	0
0.79D+00			0.96D+00			1	

PGAPack			PDS			optimal	
31 bits for the gray-coding			tol = 1.0D-08; x_0 = (-1.2,1)				
best x	best f	# iter.	best x	best f	# iter.	x_*	f_*
0.49D+00	2.55D-01	100	0.99D+00	5.90D-06	100	1	0
0.24D+00			0.99D+00			1	

DeJong test function F3 (population/iteration = 100)

$$f_3(x) = \sum_{i=1}^{5} \text{integer}(x_i), \qquad -5.12 \leq x_i \leq 5.11$$

PGAPack			PDS			optimal	
10 bits for the gray-coding			tol = 1.0D-03; x_0 = (5.11,. . . ,5.11)				
best x	best f	# iter.	best x	best f	# iter.	x_*	f_*
-5.03D+00	-2.70D+01	100	-5.01D+00	-3.00D+01	40	inf.	-30
-4.68D+00			-5.01D+00			many	
-4.04D+00			-5.01D+00				
-4.69D+00			-5.01D+00				
-5.03D+00			-5.01D+00				

DeJong test function F3 (population/iteration = 100)

$$f_3(x) = \sum_{i=1}^{5} \mathsf{integer}(x_i), \qquad -5.12 \le x_i \le 5.11$$

PGAPack			PDS			optimal	
31 bits for the gray-coding			tol = 1.0D-08; x_0 = (5.11,. . . ,5.11)				
best x	best f	# iter.	best x	best f	# iter.	x_*	f_*
-4.32D+00 -5.02D+00 -5.09D+00 -4.20D+00 -5.02D+00	-2.80D+01	100	-5.01D+00 -5.01D+00 -5.01D+00 -5.01D+00 -5.01D+00	-3.00D+01	57	inf. many	-30

DeJong test function F5 (population/iteration = 100)

$$f_5(x) = 0.002 + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2} (x_i - a_{ij})^6}, \qquad -65.536 \le x_i \le 65.536$$

PGAPack			PDS			optimal	
17 bits for the gray-coding			tol = 1.0D-04; x_0 = (50,50)				
best x	best f	# iter.	best x	best f	# iter.	x_*	f_*
64.73D+00 -65.40D+00	2.00D-03	100	65.53D+00 65.53D+00	2.00D-03	19	multiple	2.00D-03

PGAPack			PDS			optimal	
31 bits for the gray-coding			tol = 1.0D-08; x_0 = (50,50)				
best x	best f	# iter.	best x	best f	# iter.	x_*	f_*
64.73D+00 -65.40D+00	2.00D-03	100	65.53D+00 65.53D+00	2.00D-03	29	multiple	2.00D-03

Modified DeJong test function F1.12 (population/iteration = 100)

$$f_1(x) = \sum_{i=1}^{12} x_i^2, \qquad -5.12 \le x_i \le 5.11$$

PGAPack			PDS			optimal	
10 bits for the gray-coding			tol = 1.0D-03; x_0 = (5.11,. . . ,5.11)				
best x	best f	# iter.	best x	best f	# iter.	x_*	f_*
0.24D+00	3.71D+00	100	-1.50D-02	3.94D-02	100	0	0
-0.87D+00			-1.50D-02			0	
0.20D+00			6.25D-04			0	
-0.13D+00			6.25D-04			0	
-0.86D+00			9.43D-02			0	
-0.20D+00			6.25D-04			0	
0.27D+00			6.25D-04			0	
-0.24D+00			-1.50D-02			0	
6.00D-02			6.25D-04			0	
1.39D+00			9.43D-02			0	
-3.00D-02			9.43D-02			0	
-0.85D+00			0.11D+00			0	

Modified DeJong test function F1.12 (population/iteration = 400)

$$f_1(x) = \sum_{i=1}^{12} x_i^2, \qquad -5.12 \le x_i \le 5.11$$

PGAPack			PDS			optimal	
10 bits for the gray-coding			tol = 1.0D-03; x_0 = (5.11,. . . ,5.11)				
best x	best f	# iter.	best x	best f	# iter.	x_*	f_*
0.73D+00	9.31D-01	100	-6.10D-07	4.32D-10	100	0	0
-0.48D+00			-6.10D-07			0	
-5.00D-02			1.46D-05			0	
0.47D+00			-6.10D-07			0	
9.00D-02			-6.10D-07			0	
-0.24D+00			-6.10D-07			0	
0.28D+00			-6.10D-07			0	
-0.31D+00			-6.10D-07			0	
0.23D+00			1.46D-05			0	
-0.31D+00			-6.10D-07			0	
-9.00D-02			-6.10D-07			0	
-0.26D+00			-6.10D-07			0	

Modified DeJong test function F1.24

$$f_1(x) = \sum_{i=1}^{24} x_i^2, \qquad -5.12 \le x_i \le 5.11$$

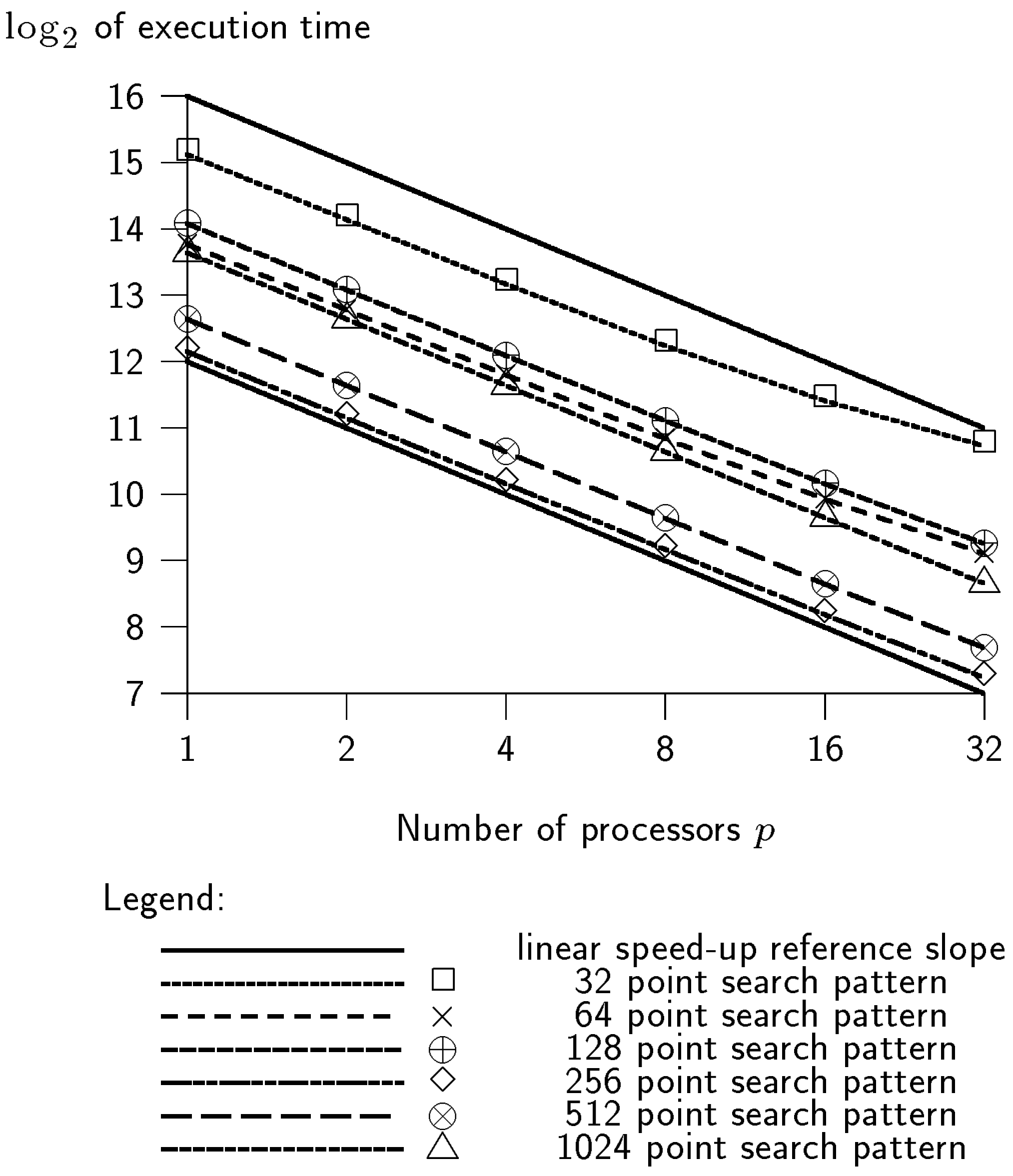
pop.	PGAPack		PDS		optimal
per	10 bits for the gray-coding		tol = 1.0D-03; x_0 = (5.11,. . . ,5.11)		
iter.	best f	# iter.	best f	# iter.	f_*
100	2.49D+01	100	2.46D+01	100	0
400	1.05D+01	100	2.46D+01	100	0
800	4.09D+00	100	1.85D+01	100	0
1600	5.10D+00	100	5.17D-03	100	0
3200	4.96D+00	100	4.72D-03	100	0

Limits of scalability for pattern search methods

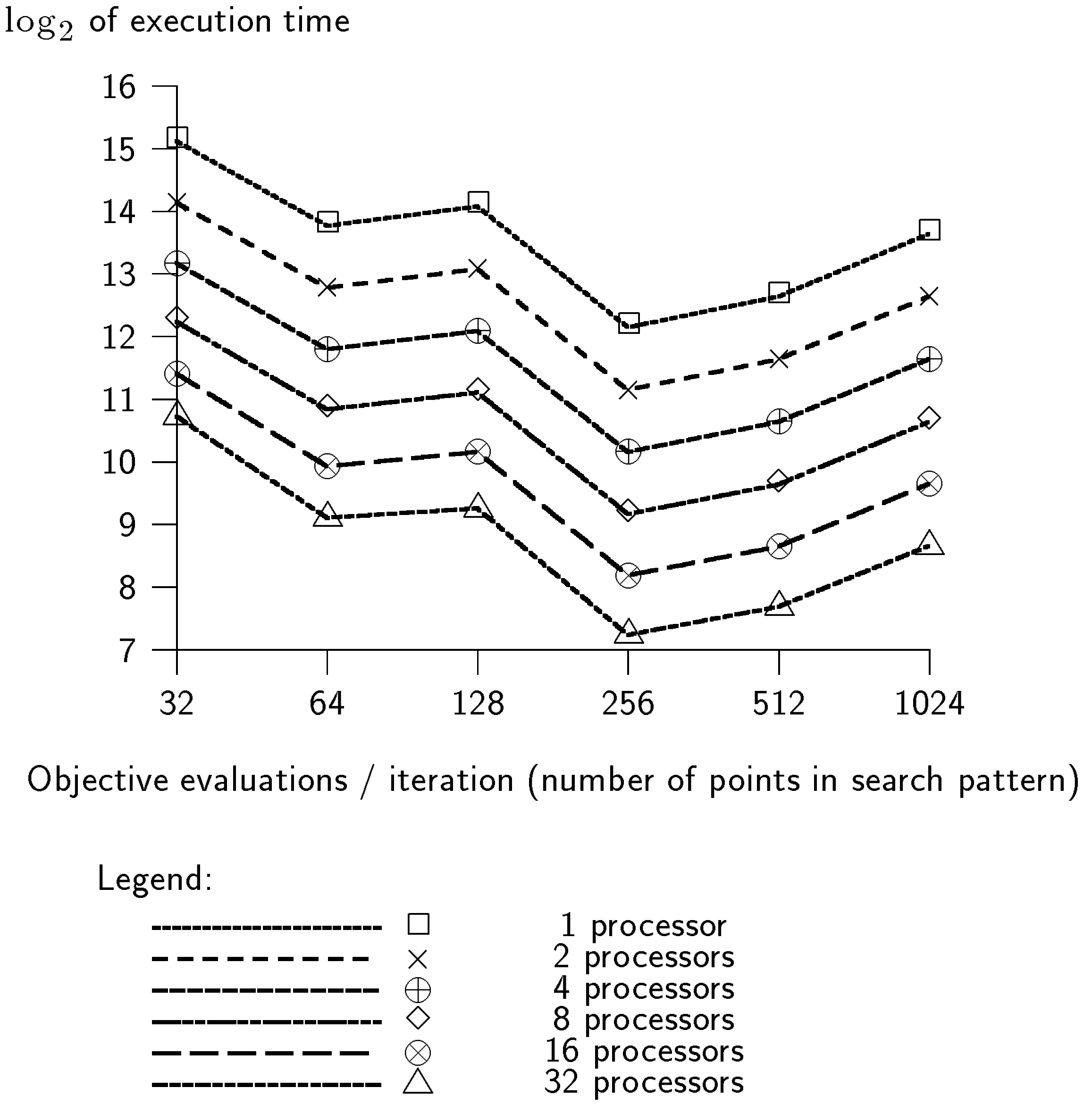
For a fixed number of points M in the pattern (i.e., fixed number of evaluations of $f(x)$ per iteration), there is a linear speed-up with p, the number of processors.

Consider, however, increasing the number of points M in the pattern while p remains fixed. This time per iteration increases as $\lceil M/p \rceil$; however, there is an unpredictable benefit to expanding the search per iteration.

Linear speed-up for pattern search as a function of p:



Nonlinear dependence of speed-up on number of objective evaluations / iteration



Methods that use derivatives

Choose $x_0 \in \mathbb{R}^n$.

For $k = 0, 1, \ldots$, until convergence do $\{$

1. Construct a linear or quadratic approximation of the nonlinear problem. This involves calculation of derivatives of the objective and constraints, Lagrange multipliers, etc.

2. Compute a trial step s_k by approximately solving the simpler approximate problem.

Methods that use derivatives

3. If $x_+ = x_k + s_k$ is acceptable, then

    set $x_{k+1} = x_+$.

   Else,

    try some manner of shorter step.

}

Because the approximate subproblem uses the Taylor's series of the objective and constraints, it reflects the behavior of the nonlinear problem for sufficiently short steps. Thus, taking shorter steps is the right response to an unacceptable step.

Unconstrained minimization

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \;\; f(x)$$

Working assumption: the cost of each of the n partial derivatives $\dfrac{\partial f}{\partial x_i}(x)$ is comparable to that of $f(x)$.

This is the case for finite-difference derivatives and the forward mode of automatic differentiation.

Unconstrained minimization: Line-search methods

Given $x_0, f(x_0), g = \nabla f(x_0)$, for $k = 0, 1, \ldots$ until convergence do $\{$

1. Construct a quadratic model of the objective:

$$f(x_k + s) \approx q(s) = f(x_k) + g^T(x_k)s + \frac{1}{2}s^T B_k s,$$

   where B_k is the Hessian or an approximation thereto.

2. Compute a step s_k by (approximately) solving $B_k s_k = -g$ or $(B_k + D)s_k = -g$ with $g^T s_k < 0$.

3. $x_+ = x_k + s_k$.

4. Compute $f(x_+)$.

Unconstrained minimization: Line-search methods

5. If $x_+ = x_k + s_k$ is acceptable

    set $x_{k+1} = x_+$ and compute $\nabla f(x_{k+1}), B_{k+1}$.

   Else,

    search along s_k, back-tracking from x_+ to x_k along s_k, until an acceptable iterate is found and update as above.

}

Unconstrained minimization: Trust-region methods

Given $x_0, f(x_0), g = \nabla f(x_0)$, until convergence do $\{$

1. Construct a quadratic model of the objective:

   $$f(x_k + s) \approx q(s) = f(x_k) + g^T(x_k)s + \frac{1}{2}s^T B_k s,$$

   where B_k is the Hessian or an approximation thereto.

2. Compute a step s_k by approximately solving

   $$\begin{array}{ll} \text{minimize} & q(s) \\ \text{subject to} & \| \, s \, \| \leq \delta_k \end{array}$$

3. $x_+ = x_k + s_k$.

4. Compute $f(x_+)$.

Unconstrained minimization: Trust-region methods

5. If $x_+ = x_k + s_k$ is acceptable

   set $x_{k+1} = x_+$, compute $\nabla f(x_{k+1}), B_{k+1}$, and update δ_k.

   Else,

   either reduce δ_k and compute a new s_k, or revert to a line-search along s_k until an acceptable iterate is found and update as above.

}

Line-searches vs. trust-regions

Line-search methods first choose a search direction and then determine the length of the step along that direction.

Trust-region first choose the length of the step and then compute the step.

Methods that use derivatives: Quasi-Newton algorithms

Advantages:

- Most efficient in terms of number of iterations.
- Large components relatively simple to parallelize.
- Can handle large numbers of variables and constraints.
- Robustness / convergence analysis.

Methods that use derivatives: Quasi-Newton algorithms

Disadvantages:

- Difficult to implement.
- Non-trivial sequential component.
- Require derivatives or good approximations thereof.
- General-purpose algorithms have limited scalability.
- More easily fooled by local minimizers and noise.

Opportunities for parallelism

1. Construction of the quadratic model.

   Parallelism in the calculation of the required derivatives.

2. Computation of the trial step s_k.

   Parallelism (possibly) in the computation of the trial step.

3. Making the sequential logic concurrent.

   To preempt bottlenecks, calculate the quantities needed for all possible paths through the sequential logic:

   - Find an acceptable step (possibly an iterative process).
   - Update as appropriate.

Calculating derivatives: finite-differences

Naive finite-differences parallelize trivially:

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + h_i e_i) - f(x)}{h_i}$$

One distributes the different finite-difference perturbations to different processors:

```
for i = 1, ..., n {   /* Parallel DO-loop */
  compute f(x + h_i e_i)
}
compute the difference quotients
```

Calculating derivatives: finite-differences

Sophisticated, adaptive finite-difference schemes (such as those in NPSOL and PORT) may from time to time perform what amounts to a line-search to find an appropriate finite-difference interval.

This can lead to bottlenecks since for some i, $f(x + h_i e_i)$ might be evaluated for more than a single value of h_i.

We can make such schemes concurrent by applying preemptive parallelism, at the cost of using more processors and altering the logic of the finite-difference algorithm.

Second derivatives

Finite-difference estimates of second derivatives can be parallelized similarly to approximate the Hessian. However, to approximate $f(x)$ and $\nabla f(x)$ requires only $n + 1$ processors, while $f(x), \nabla f(x)$ and $\nabla^2 f(x)$ requires $(n^2 + 5n + 2)/2$.

For even moderate n, it is likely that the number of processors p lies between $n + 1$ and $(n^2 + 5n + 2)/2$.

Partial approximation of the Hessian can be used in conjunction with secant updates. The Hessian estimate used to compute s_k involves both the available partial second derivatives and a secant update; only the finite-difference approximation carries forward.

Byrd, Schnabel, and Shultz found this approach promising: not linear speed-up, but definitely better than a pure secant update code.

Calculating derivatives: automatic differentiation

Automatic differentiation (AD) is a source-to-source translation of a program (e.g., C or Fortran) to another program that has the functionality of the original program and also computes derivatives of specified outputs with respect to specified inputs.

AD is **not** symbolic differentiation. AD is based on a statement-by-statement application of the chain rule.

One such tool for Fortran, ADIFOR, was developed at the CRPC (Bischof, Carle, et al.).

Automatic differentiation

Advantages:

- AD is automatic.
- AD can be more reliable than finite-differences.

Shortcomings:

- AD output is hard to read.
- AD can inhibit vectorization and other compiler optimizations.
- AD might not understand parallelism.
- Second derivatives are not easily available.

Automatic differentiation

Features:

- Cost of $\partial f/\partial x_i$ comparable to that of $f(x)$ for the forward mode of AD.

- Reverse mode of AD requires fewer flops but more memory access.

- AD can differentiate non-differentiable processes.

ADIFOR input: dot product

```
      subroutine dot (xy, x, y, n)
c
c****************************************
c
      integer n, i
      real x(n), y(n), sum
c
c****************************************
c
      sum = 0.0
c
      do 120 i = 1, n
        sum = sum + x(i) * y(i)
  120 continue
c
      xy = sum
c
      return
      end
```

ADIFOR output: dot product

```
      subroutine g_dot(g_p_, xy, g_xy, ldg_xy,
     >                 x, g_x, ldg_x, y, n)
C
C******************************************
C
      integer n, i
      real x(n), y(n), sum
C
C******************************************
C
      integer g_pmax_
      parameter (g_pmax_ = 1000)
      integer g_i_, g_p_, ldg_x, ldg_xy
```

```
      real g_sum(g_pmax_), g_x(ldg_x, n), g_xy(ldg_xy)
      save g_sum
      integer g_ehfid
      data g_ehfid /0/
C
      call ehsfid(g_ehfid, 'dot','g_dot.f')
C
      if (g_p_ .gt. g_pmax_) then
        print *, 'Parameter g_p_ is greater than g_pmax_'
        stop
      endif
      do g_i_ = 1, g_p_
        g_sum(g_i_) = 0.0
      enddo
      sum = 0.0
C
```

```
      do 99999 i = 1, n
        do g_i_ = 1, g_p_
          g_sum(g_i_) = g_sum(g_i_) + y(i) * g_x(g_i_, i)
        enddo
        sum = sum + x(i) * y(i)
120     continue
99999 continue
C
      do g_i_ = 1, g_p_
        g_xy(g_i_) = g_sum(g_i_)
      enddo
      xy = sum
C
      return
      end
```

AD and parallel computation: derivative strip-mining

Code generated by ADIFOR can be parallelized:

```
      /*
       * Distribute chunks of the g_i_
       * index range to different processors.
       */

      do 99999 i = 1, n
        do g_i_ = 1, g_p_
          g_sum(g_i_) =
     >      g_sum(g_i_) + y(i) * g_x(g_i_, i)
        enddo
        sum = sum + x(i) * y(i)
99999 continue
C
      do g_i_ = 1, g_p_
        g_xy(g_i_) = g_sum(g_i_)
      enddo
      xy = sum
```

AD and parallel computation: derivative strip-mining

This is technique, called derivative strip-mining, is facilitated by the use of the input "seed matrix."

Applied to differentiating a set of outputs $F(x) = (F_1(x), \ldots, F_m(x))$, ADIFOR produces code to compute

$$DF * S,$$

where DF is the Jacobian of F.

By varying the input "seed matrix" S, the output will be different columns of DF. If $F(x) = f(x)$ is a scalar, we obtain subsets of partial derivatives.

This allows us to distribute the calculations in the code generated by ADIFOR.

Benefits of parallelization

Let

t_f = the time to evaluate $f(x)$,

β = the average number of additional objective evaluations due to unsuccessful steps (i.e., $1 + \beta$ is the average number of steps per iteration that must be evaluated to find an acceptable step),

p = the number of processors available.

Assume that we use a line-search method with

1. finite-difference 1st derivatives, and
2. approximate the Hessian with a secant update (e.g., BFGS),

and that the cost of solving for the search direction is negligible.

The sequential algorithm costs

$$((1+\beta)+n) * t_f$$

objective evaluations ($\propto$ run-time) per iteration.

If we parallelize the calculation of 1st derivatives, the cost per iteration is

$$\left((1+\beta)+\left\lceil\frac{n}{p}\right\rceil\right) * t_f.$$

The quantity $1+\beta$ is now the limit on parallel speed-up.

Preemptive computation to avoid sequential logic

The term $1+\beta$ is due to the sequential logic of the iterative process of finding an acceptable new iterate x_+ and updating:

If the step $x_+ = x_k + s_k$ is acceptable $\{$

compute $\nabla f(x_+)$ via n parallel finite-difference perturbations

$\}$

Else $\{$

backtrack from x_+ to x_k along s_k; this requires computing $f(x_k + \alpha s_k)$ for one or more values of $\alpha \in (0,1)$.

$\}$

Processors sit idle until a step is accepted.

We can avoid this sequential bottleneck by using the parallelism for preemptive computation.

Preemptive computation to avoid sequential logic

Parallel logic: go ahead and compute the quantities needed in the event of both success and failure:

1. $\nabla f(x_+)$, and (possibly)

2. some number of points along the search direction s_k in the event of back-tracking/trust region reduction.

Option 1, preemptive computation of $\nabla f(x_+)$ leads to a parallel per iteration cost of

$$\left(\beta + \left\lceil \frac{n+1}{p} \right\rceil\right) * t_f.$$

Parallel line searches

What about option 2, parallelizing the line search?

The usual line-search with safe-guarded backtracking is inherently sequential:

1. If $x_+ = x_k + s_k$ is unacceptable, construct a quadratic model of $\phi(\alpha) = f(x_k + \alpha s_k)$ from $\phi(0) = f(x_k)$, $\phi'(0) = \nabla f(x_k)^T s_k$, and from $\phi(1) = f(x_+)$.

2. Compute $x'_+ = x_k + \alpha' s_k$ from the quadratic model.

3. If x'_+ is unacceptable, use the new datum $f(x'_+)$ to construct a cubic model.

4. Compute a new trial step using the cubic model.

5. Etc.

Parallel line searches

A parallel line-search might compute $f(x)$ at a number of points along $x_k + \alpha s_k$ concurrently in the hopes that an acceptable x_+ might be found without iteration.

As long as x_+ satisfies the Armijo-Goldstein-Wolfe sufficient descent direction, all is well.

Note that this approach will produce iterates that might not be the same as those produced by the sequential algorithm.

Similar considerations hold for trust regions.

Limits to speed-up via trying to preempt backtracking or reducing the trust radius

A line-search or trust region step requires about 1.25–1.6 objective evaluations per iteration (i.e., we accept the first x_+ we try 60–80% of the time). (YMMV, obviously.)

This means β, the average number of additional function evaluations for unsuccessful steps, is typically in the range 0.25–0.6. This limits the parallel speed-up we should expect from removing the sequentiality from the process of finding an acceptable step:

$$\left(\beta + \left\lceil \frac{n+1}{p} \right\rceil\right) * t_f.$$

However, there exist problems for which this approach may prove fruitful.

Automatic differentiation and preemptive computation

There is an additional subtlety if we preemptively compute $\nabla f(x_+)$ using automatic differentiation.

The forward mode of AD generates code to compute $\partial f / \partial x_i$ in time $\mu_{AD} * t_f$, where $1 \leq \mu_{AD} \leq 2$.

This means that if x_+ is found to be unacceptable, the processors computing $\nabla f(x_+)$ may continue to compute unless signaled to stop.

AD introduces a dynamic task management element to the parallel preemptive computation.

What about supererogatory computation?

A number of people (e.g., Byrd, Schnabel and Shultz; Dixon; Nash and Sofer) have looked at accelerating the progress of the optimization by considering additional points as the next iterate.

E.g., one can compute $f(x)$ at many points along the usual search direction s or in a cone or tube around s in the hope of finding a better next iterate.

Though this approach puts any idle processors to work, it has not been found to improve the performance of the sequential algorithms—the usual step s_k is "too good."

Moreover, the Broyden-Dennis-Moré theory tells us that after some iteration we should expect s to be always acceptable: asymptotically, the expected cost of the search is 1 objective evaluation.

$\Rightarrow$ Asymptotically, no benefit from this use of parallelism.

Computing the trial step

Depending on the number of optimization variables, this might or might not be worth parallelizing.

- Parallel direct linear algebra (sparse Cholesky).
- Truncated Newton methods (sparse/dense).

Truncated Newton methods

These methods are useful where the Hessian is large and either sparse or dense, and cannot be practically assembled (e.g., in reduced basis methods for constrained minimization).

These methods use iterative linear algebra (e.g., Lanczos or conjugate gradients), so only matrix-vector products are needed.

Iterative linear algebra parallelizes easily (though preconditioning might not).

Truncated Newton methods can be viewed as interpolating between the steepest descent direction and the Newton direction.

Truncated Newton line-search methods

Let $g_k = \nabla f(x_k)$.

Dembo, Eisenstat, and Steihaug: Approximately solve the Newton system

$$B_k s_k = -g_k$$

Nash, Nash and Sofer: Apply Lanczos to form and solve a modified positive definite Newton system

$$(B_k + D)s_k = -g_k$$

The BTN (Block Truncated Newton) of Nash and Sofer is a parallel implementation of a block Lanczos truncated Newton algorithm.

Truncated Newton trust-region methods

Steihaug-Toint: Apply a conjugate direction method to reduce the quadratic model subject to the trust region constraint

$$\begin{array}{ll} \text{minimize} & g_k^T s + \frac{1}{2} s^T B_k s \\ \text{subject to} & \| s \| \le \delta_k \end{array}$$

Proceed until one of the following occurs:

1. the iterate leaves the trust region,

2. a direction of negative curvature is found,

3. the Newton step is found,

4. patience is exhausted.

Constrained minimization

The opportunities for parallelism are much the same as those for unconstrained minimization.

Constrained minimization

Reduction to unconstrained minimization.

- Exact penalty methods.
- Successive unconstrained minimization (SUMT).
  - Classical and modified barrier methods.

Constrained minimization

Reduction to equality constrained minimization with simple bounds.

- Augmented Lagrangian methods.

This, in turn, leads to a bound constrained minimization problem.

Constrained minimization

Successive approximations of the NLP.

- Successive linear programming (SLP).
- Successive quadratic programming (SQP).
- Active set methods.
- Feasible direction methods.

Successive approximations

In SLP and SQP, the main computational cost is that of constructing and solving the subproblem, which is either an LP or a QP. There may be a possibility for parallelism in the solution of the subproblem.

Active set methods reduce the problem to unconstrained minimization in subspaces. One of the most successful general-purpose NLP codes, the NPSOL code of Gill, Murray, Saunders, and Wright, is an active set method.

Feasible direction methods require the solution of an LP.

Parallelism in constrained optimization

The opportunities for parallelism are much the same as those in unconstrained optimization.

Constrained optimization has a number of additional sequential components (e.g., multiplier estimates, active set selection).

Parallelism in constrained optimization

Constraint derivatives can be computed in parallel.

There may not be much speed-up in parallelizing the evaluation of the individual constraint components $g(x) = (g_1(x), \ \ldots, \ g_m(x))$.

The constraints are often fairly cheap, or they are expensive but coupled:

$$g(x, u(x)) \geq 0$$

where $u(x)$, say, might be the solution of a PDE.

In such cases, computing $u(x)$ is the bottleneck, not $g(x, u(x))$.

Parallelism in nonlinear optimization: summary

Recall that:

n = number of optimization variables,

m = number of constraints,

p = number of processors.

Genetic algorithms

> **Performance:** Performance decreases as m and n increase.
>
> **Scalability:** Performance increases as p increases, without a limit.
>
> However, the scalability is due to the fact that the basic algorithm is inefficient (i.e., other less expensive algorithms are superior).

Parallelism in nonlinear optimization: summary

Pattern search methods

Performance: Performance decreases as m and n increase, though not as badly as with GAs.

Scalability: Performance increases as p increases, to some unpredictable point.

Quasi-Newton methods

Performance: Performance is relatively stable as m and n increase (though the cost per iteration increases with m and n).

Scalability: Scalability is a function primarily of n. We can use $O(n)$ processors if we compute only 1st derivatives and $O(n^2)$ if we estimate all or part of the 2nd derivatives.

Global optimization

Operating assumptions: global (nonlinear, nonconvex) optimization

- the objective and/or constraints is/are expensive to evaluate,
- the first derivatives also are expensive to evaluate,
- the number of optimization variables n and constraints m is relatively small, and
- a large number of evaluations ($\gg n$) may required at each iteration and there may be a large number of iterations k.

The implication is then that the expensive portion of the calculation lies in the evaluation of the objective, constraint, and gradient values, and *not* in the linear algebra.

Note: coarse granularity; a high degree of concurrency.

Space changes with increasing dimension

One way to measure how space ($I\!R^n$) changes as the dimension n increases is to look at the relationship between the ℓ^2 and ℓ^∞ norms: for $x = (x_1, \ldots, x_n)$,

$$\begin{aligned} \| x \|_{\ell^2} &= \left(\sum_{i=1}^{n} x_i^2 \right)^{\frac{1}{2}} \\ \| x \|_{\ell^\infty} &= \max_{1 \le i \le n} | x_i | \end{aligned}$$

and

$$\| x \|_{\ell^\infty} \le \| x \|_{\ell^2} \le \sqrt{n} \, \| x \|_{\ell^\infty} .$$

Space becomes sparser with dimension: The curse of dimensionality

Another measure is related to volume. Consider the unit ℓ^∞ ball (hypercube) $\Omega \subset I\!R^n$.

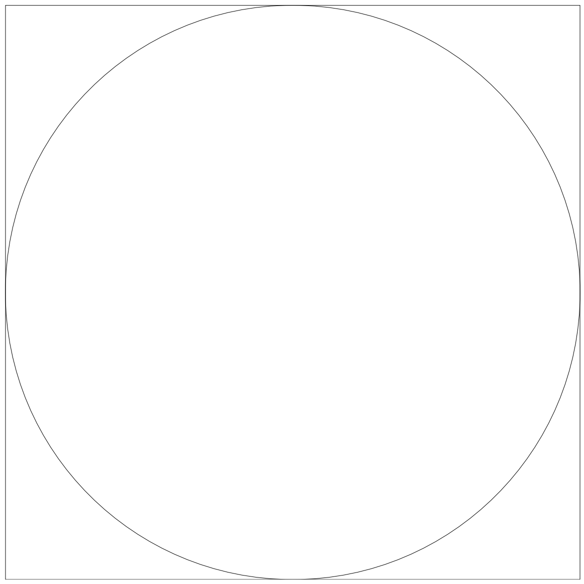
Suppose we want a sampling of points S in the unit hypercube that insures for any point $x \in \Omega$ we have a point $z \in S$ such that $\| x - z \|_{\ell^\infty} \leq 0.1$. How many points do we need in S?

Answer: 10^n.

Uh-oh—for $n = 2, 3$ this isn't so bad, but $n = 100$ is not a large optimization problem, while 10^{100} is a relatively large number (e.g., $10^{100} \gg$ than the estimated number of atoms in the universe).

The curse of dimensionality

Consider also the relative volumes of the unit ℓ^2 and ℓ^∞ balls:

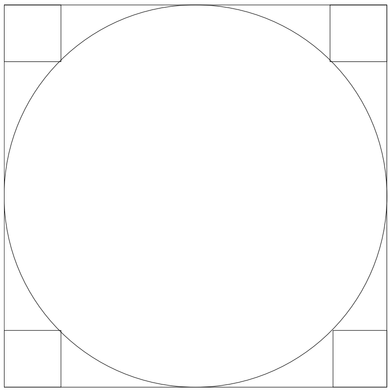


$$\frac{\text{volume of unit } \ell^2 \text{ ball}}{\text{volume of unit } \ell^\infty \text{ ball}} = \frac{\pi^{n/2}}{n 2^{n-1} \Gamma(n/2)}$$

The curse of dimensionality

Dimension	n=2	n=3	n=5	n=7	n=100
Ratio	0.785	0.524	0.164	0.037	$\sim 10^{-69}$

Most of the volume of the cube is not in its center.



Volume of disjoint corner cubes: $2^n \left(\frac{\sqrt{n} - 1}{\sqrt{n}} \right)^n$.

For $n = 100$, the 2^{100} corners account for $< 3 \times 10^{-5}$ of the cube: Most of the volume is not in the corners, either.

The curse of dimensionality

Techniques developed for high-dimensional numerical integration provide some relief to the problem of dimension in sampling.

However: Do not expect anything (including parallelism) to make general global optimization practical if the objective and/or constraints are expensive and there are more than a few variables.

Global optimization

We will discuss two broad categories that lend themselves to parallelization:

- Bounding methods
- Stochastic methods

Bounding methods

Bounding methods include:

- Branch and bound
- Interval methods
- Lipschitz methods

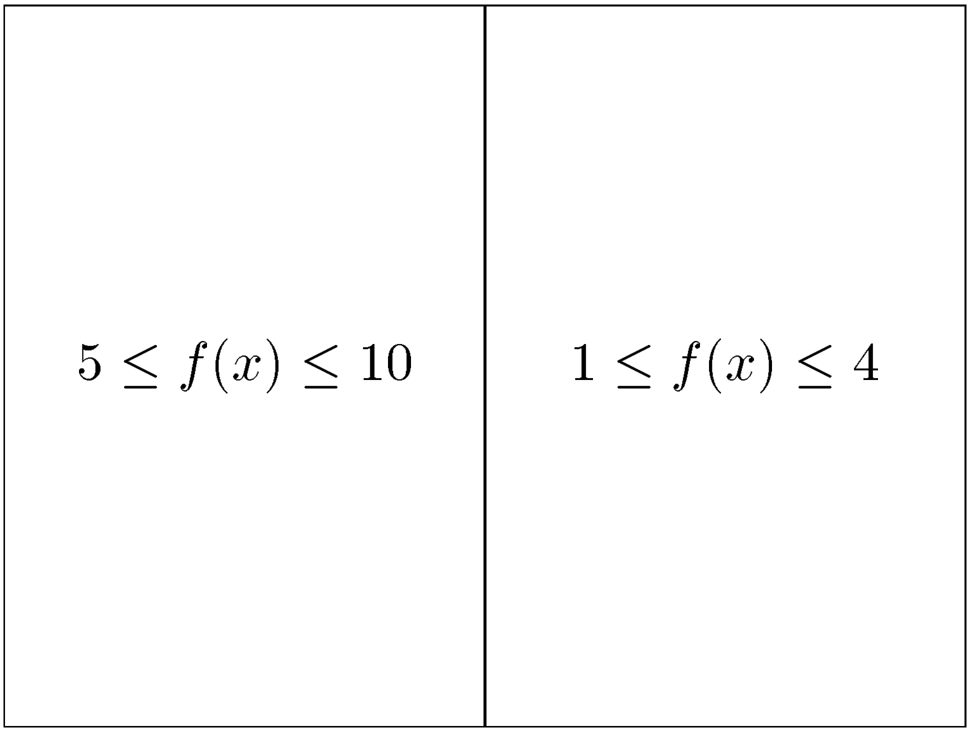
Bounding methods

These proceed as follows:

1. Partition the initial search space into smaller subregions,
2. Bound the objective (and possibly its derivatives) over the subregions, and
3. Use the bounds calculated in step 2 to eliminate those subregions which cannot contain a global minimizer and return to step 1.

The partitioning can be parallelized.

Bounding methods



This partition reveals that the global minimizer must lie in the subregion on the right. We would then partition that subregion and continue.

Bounding methods

Advantages:

- The suboptimality of the computed solution can be estimated.

Shortcomings:

- Bounding methods suffer from the curse of dimensionality.
- Constraints other than simple bounds on the variables make the partitioning problematic.
- Suboptimality estimates may be overly pessimistic leading to excessive computation.

Lipschitz methods

Suppose that for any feasible x, y,

$$| f(x) - f(y) | \leq K \| x - y \| .$$

The smallest admissible K is called the Lipschitz constant.

As we partition the search space, we can bound f on the subregions using this estimate, provided we have an (over-)estimate of K: If we know $f(x)$, then

$$f(x) - \varepsilon \leq f(y) \leq f(x) + \varepsilon$$

for all y within distance ε / K of x.

Problems: Large value (or estimate) of K; high dimension.

Interval methods

For a given range of inputs we can determine the range of the outputs of the various elementary operations that comprise numerical calculations (i.e., computer codes).

Interval methods track the inputs and outputs through the calculation to obtain a bound on how large or small the objective can be on a given subregion.

This can be accomplished through automatic analysis of the computer code.

Mathematically the approach is similar to Lipschitz methods, with the same shortcomings.

Stochastic methods

These rely on sufficient rather than exhaustive exploration of the feasible region.

1. Multi-start methods

2. Genetic algorithms

The behavior of multi-start methods can be analyzed probabilistically. No general statements can be made for genetic algorithms.

Multi-start methods

1. Generate a number of random points in Ω.
2. Start a local minimization algorithm from each.
3. Choose the lowest local minimizer found.
4. Repeat.

A good multi-start method (e.g., Rinnooy Kan and Timmer) carefully selects its start points to be sufficiently far apart to reduce or eliminate finding the same local minimizer by multiple local minimizations.

For such methods one can show that one finds with probability 1 all the local minimizers in a finite number of iterations.

Moreover, if one iterates forever the number of local minimization (step 2) remains finite with probability 1 (i.e., asymptotically we do not needlessly revisit points).

Parallel multi-start methods

```
until convergence do {
  generate a set of random points X in Ω
  for each x ∈ X { /* Parallel tasks */
    run a local minimization algorithm from x.
  }
  choose the lowest local minimizer found.
}
```

The local minimizations are asynchronous since they will run for different intervals.

Elaborations involving dynamic, adaptive partitioning of the search space followed by the creation of more starting points can make multi-start algorithms much more effective (at the cost of making the control of the tasks more complicated in the parallel setting).

Decomposition ideas

The basic decomposition idea

At each iteration:

1. Relax coupled blocks to obtain a block-separable optimization problem.

2. Solve (to some extent) the independent optimization problems.

3. Synthesize a step for the original optimization problem from the solutions of the decoupled problems.

   This is a sequential step.

Linear programming: The Dantzig-Wolfe decomposition

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax = b \\ & x \geq 0 \end{array}$$

Constraint matrix:

$$A = \begin{pmatrix} B_1 & B_2 & \dots & \dots & B_p \\ A_{11} & 0 & \dots & \dots & 0 \\ 0 & A_{22} & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & 0 & A_{pp} \end{pmatrix}$$

If the B_i were absent, the LP would decouple.

Linear programming: The Dantzig-Wolfe decomposition

At each iteration:

1. Relax the coupling to obtain a collection of independent LPs involving the A_{ii} (and some data reflecting the coupling).

2. Solve the resulting independent LPs.

3. Synthesize a step for the original LP from the solutions of the decoupled LPs. This is a sequential step.

The sequential Dantzig-Wolfe decomposition was never found to be effective (solving the original LP is more efficient).

Parallelization of the Dantzig-Wolfe decomposition is also ineffective (Ho, et al.) because of the limited parallelism and the sequential synthesis step.

General decomposition of nonlinear problems

Ferris & Mangasarian:

$$\underset{x \in \Omega}{\text{minimize}}\, f(x)$$

where Ω is convex.

The idea is to apply decomposition at each iteration to find a good subspace for a sequential optimization.

For p processors, partition x into p blocks $x^1, \ldots, x^p$. For a block x^i, let $\widehat{x^i}$ denote the complementary variables.

1. Decomposition: For $i \in \{1, \ldots, p\}$, distribute the parallel solution of

   $$\begin{array}{ll} \underset{x^i}{\text{minimize}} & f(x^i, \hat{x}^i_k) \\ \text{subject to} & (x^i, \hat{x}^i_k) \in \Omega \end{array}$$

   for a solution $z^i = (x^i, \hat{x}^i_k)$.

2. Synchronization: Synthesize a new trial iterate for the original NLP by minimizing $f(x)$ over the affine hull of the subproblem solutions z^i

$$
\begin{array}{ll}
\text{minimize} & f(x) \\
\text{subject to} & x = \mu_0 x_k + \sum_{i=1}^{p} \mu_i z^i \in \Omega \\
& \sum_{i=0}^{p} \mu_i = 1
\end{array}
$$

for x_+ and check acceptability.

This is a sequential step.

Elaborations were considered in which the subproblems involved some information from the complementary variables (i.e., coupling).

The value of this approach is not clear. Ferris & Mangasarian reported results only for unconstrained problems, and did not compare the performance of the decomposition approach with a conventional sequential algorithm.

The speed-up due to distributing the decomposition onto increasing numbers of processors was far less than linear due to the significant sequential component of the synchronization step.

Also note that as x is decomposed into more blocks (e.g., if a large number of processors were available), the subproblems become smaller while the sequential synchronization problem becomes larger. This means the sequential component grows while the parallel component shrinks, potentially a severe limit on scalability.

Decomposition of state equations

Dennis and Lewis looked at introducing concurrency to nonlinear optimization problems via decomposition of state equations and other coupled systems.

Consider

$$\text{minimize } F(x) = f(x, u(x))$$

where $u(x)$ is computed by solving an implicit relation $h(x, u(x)) = 0$ (e.g., a differential equation or engineering analysis). This solution for u is assumed to be an expensive sequential step.

Decomposition of state equations: two block example

Suppose we can write h as

$$h(x,u) = h(x,u_1,u_2) = \left(\begin{array}{c} h_1(x,u_1,u_2) \\ h_2(x,u_1,u_2) \end{array} \right)$$

where

1. $h_1(x,u_1,u_2) = 0$ can be solved for u_1 given x and u_2, and

2. $h_2(x,u_1,u_2) = 0$ can be solved for u_2 given x and u_1.

We also assume that h_2 is block-diagonal in u_2. E.g., if h were linear in u,

$$h(x, u_1, u_2) = \begin{pmatrix} A_{11}(x) & A_{12}(x) \\ A_{21}(x) & A_{22}(x) \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$$

where

$$A_{22} = \begin{pmatrix} C_{11} & 0 & \dots & \dots & 0 \\ 0 & C_{22} & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & 0 & C_{pp} \end{pmatrix}$$

Such structure can be imposed by applying domain decomposition to ODE and PDE; it also arises naturally in coupled engineering systems.

Decomposition of state equations

We then reformulate the optimization problem as

$$\begin{array}{ll} \text{minimize} & F(x, u_1) = f(x, u_1, u_2(x, u_1)) \\ \text{subject to} & h_1(x, u_1, u_2(x, u_1)) = 0 \end{array}$$

where we obtain $u_2(x, u_1)$ via the solution of

$$h_2(x, u_1, u_2) = 0,$$

which is block-diagonal in u_2 and thus can be solved in parallel.

The granularity depends on the coarseness of the decomposition of h_2.

Decomposition of state equations

The preliminary results are promising. Sometimes the decomposition approach is much more efficient than a standard approach, particularly for hard problems. Often the performance is about the same, possibly because the test problems (parameter estimation for elliptic PDE) are too easy.

The equality constrained minimization algorithm that is used couples the independent subproblems in a way that probably degrades the efficacy of the approach; it definitely reduces concurrency.

Decomposition in the engineering optimization literature

- Concurrent subspace optimization (Sobieski, et al.)

- Collaborative optimization (Braun, Kroo, et al.)

It is not clear how well these methods work.

Optimization software

Caveat

Much of the extant work (particularly in nonlinear optimization) was done in the period 1984–1992, and the machines that were used (Intel iPSC 1, iPSC 2, Thinking Machines CM-2, Sequent Balance and Symmetry) have very different latency and scalar performance properties than current parallel machines.

Keep this in mind when reviewing the literature.

Parallel optimization software

- pPCx—parallel interior point code (Coleman, et al.)
- Parallel branch and cut (Trotter, et al.)
- PGAPACK—parallel genetic algorithm (Levine).
- PDS—parallel multidirectional search (Torczon).
- BTN—block truncated Newton (Nash and Sofer).

Sequential optimization software

For instance...

- AT&T PORT library (Gay). Handles only simple bounds.
- Harwell (Powell and others).
- LANCELOT (Conn, Gould, Toint). Augmented Lagrangian and bound constrained trust region methods.
- MINPACK (Moré et al.)
- NPSOL (Gill, Murray, Saunders, Wright).

See the compendium by Moré and Wright.

Optimization Technology Center at Argonne National Laboratory.