GPCG: A Case Study in the Performance and Scalability of Optimization Algorithms*

Steven J. Benson, Lois Curfman McInnes, and Jorge J. Moré

Mathematics and Computer Science Division
Argonne National Laboratory

The Toolkit for Advanced Optimization (TAO) focuses on the design of large-scale optimization software, including nonlinear least squares, bound-constrained optimization, and nonlinearly constrained optimization. The solution of optimization problems pervades many areas of computational science and demands robust and flexible solution strategies. Since many application problems require the computational power of high-performance computers, a need clearly exists for a uniform and flexible framework for developing software for solving large-scale optimization problems.

The TAO design philosophy uses object-oriented techniques of data and state encapsulation, abstract classes, and limited inheritance to create a flexible optimization toolkit. The algorithms in the toolkit place strong emphasis on the reuse of external tools where appropriate. Our design enables bidirectional connection to lower level linear algebra support (for example, parallel sparse matrix data structures) provided in toolkits such as PETSc and ISIS++, as well as higher level application frameworks such as POOMA and InDEPS. Our design decisions are strongly motivated by the challenges inherent in the use of large-scale distributed memory architectures and the reality of working with large, often poorly structured legacy codes for scientific applications.

Implementation of GPCG

One component within TAO is the GPCG algorithm for solving the optimization problem

$$\min\{q(x) : l \leq x \leq u\},$$

where $q : \mathbb{R} \mapsto \mathbb{R}$ is a strictly convex quadratic, and the vectors l and u are lower and upper bounds on the variables. The GPCG algorithm was originally designed by Moré and Toraldo for large-scale problems but until this work within TAO had only been implemented for a single processor. Our implementation of GPCG has been used to solve problems with over 2.5 million variables on 64 processors of the IBM SP at Argonne National Laboratory.

The GPCG uses a gradient projection algorithm to predict the face where the solution lies, and a conjugate gradient algorithm to explore this face. Thus, at each iteration the GPCG algorithm must use a conjugate gradient algorithm to solve (approximately) a system of linear equations whose coefficient matrix is the submatrix of the Hessian of q with respect to the free variables for the current iterate. Since the free variables can change arbitrarily

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

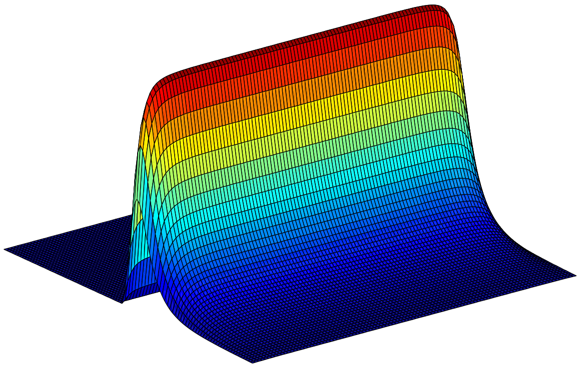

Figure 1: The journal bearing problem with $\varepsilon = 0.9$.

from iterate to iterate, solving these systems of linear equations may create a potential load-balancing bottleneck that must be addressed in an efficient implementation.

The current implementation uses the parallel system infrastructure and linear algebra offered by PETSc, which uses MPI for all interprocessor communication. The PETSc package supports the vector, matrix, index set, and linear solver objects. Functional support for the vector class includes the creation, duplication, and destruction of the objects, operations that copy elements from one vector to another, set the elements equal to a specified scalar, as well as functions for basic linear algebra operations.

The implementation of GPCG uses abstractions for vectors and matrices and hence makes no assumptions about the representation of data within them. This approach eliminates some of the barriers in using independently developed software components because it employs data-structure-neutral vector and matrix objects within the optimization algorithm and calls external libraries for the execution of linear algebra utilities.

Computational Results

We have evaluated the performance of the GPCG implementation on a variety of architectures. The data presented in this section was generated on the IBM SP (each processor has 256 MB RAM, 128 KB cache for data, and a 32 KB cache for instructions) at Argonne National Laboratory.

As a benchmark application we have used a journal bearing model, a variational problem over a two-dimensional region. Since we are interested in large problems, a grid with 1600 points in each direction was used, leading to a problem with $2.56 \cdot 10^6$ variables.

The journal bearing formulation depends on the eccentricity parameter $\varepsilon \in (0, 1)$. This parameter influences, in particular, the difficulty of the problem. Figure 1 shows the solution of the journal bearing problem for $\varepsilon = 0.9$. The steep gradient in the solution makes this problem a difficult benchmark.

Table 1 presents performance data for GPCG. We show the number of processors p, the number of GPCG iterates (faces), the number of conjugate gradient iterations n_{CG}, the wall clock solution time (in seconds), the percentage of time ($t_{CG}\%$) used by the conjugate gradient algorithm, and the efficiency ($\mathcal{E}$) of GPCG.

ε	p	faces	n_{CG}	time	$t_{CG}\%$	$\mathcal{E}$
0.1	8	46	502	3341	89	100
0.1	16	45	486	1781	89	94
0.1	32	46	480	928	89	90
0.1	64	46	504	522	89	80
0.9	8	37	102	1407	94	100
0.9	16	40	100	816	94	86
0.9	32	41	100	392	93	90
0.9	64	40	102	217	94	81

Table 1: Performance of GPCG on the journal bearing problem with $2.56 \cdot 10^6$ variables.

The results in Table 1 are noteworthy is several ways. First of all, the number of faces visited by GPCG is remarkably small. Other strategies can lead to a large number of gradient projection iterates, but the GPCG algorithm is remarkably efficient. Another interesting aspect of these results is that due to the low memory requirements of iterative solvers, we were able to solve these problems with only $p = 8$ processors. Strategies that rely on direct solvers are likely to need significantly more storage, and thus more processors. Finally, these results show that the GPCG implementation has excellent efficiency. For example, the efficiency of GPCG with respect to $p = 8$ processors ranges between 80% and 90% when $\varepsilon = 0.9$. This sustained efficiency is remarkable since the GPCG algorithm is solving a sequence of linear problems with a coefficient matrix set to the submatrix of the Hessian of q with respect to the free variables for the current iterate. Thus, our implementation's repartitioning of submatrices effectively deals with the load-balancing problem that is inherent in the GPCG algorithm.

An important aspect of our results that is not apparent from Table 1 is that for these results we were able to experiment easily with all the preconditioners offered by PETSc. In particular, we were able to compare the diagonal Jacobi preconditioner with block Jacobi and overlapping additive Schwarz preconditioners that use a zero-fill ILU solver in each block. We also experimented with a parallel zero-fill incomplete Cholesky preconditioner provided by a PETSc interface to the BlockSolve95 package of Jones and Plassmann. Interestingly enough, the diagonal Jacobi preconditioner achieved better performance on this problem.

The ability to experiment with various preconditioners is a direct result of our design philosophy that allows for bidirectional connection to the linear algebra infrastructure provided in toolkits such as PETSc and ISIS++. In this spirit, we plan to experiment with the evolving interfaces under development by the Equation Solver Interface (ESI) and Common Component Architecture (CCA) working groups, with a goal of enabling dynamic use within TAO of any ESI-compliant preconditioning components.