from *SIAM News*, Volume 32, Number 4

Exploiting Parallel Hardware in Solving Optimization Problems

By Erling Andersen and Knud Andersen

Making an optimal choice, given a number of possibilities and constraints, is a fundamental problem in economics and engineering. Examples include the optimal choice of a portfolio of stocks, given a budget and diversity requirements; the optimal design of a truss, given certain load requirements; and the determination of an optimal production plan, given the current manufacturing equipment.

In many cases such problems can be formulated as mathematical optimization problems, in which a function is maximized or minimized, given a set of decision variables and mathematical constraints specifying the set of feasible decisions. Efficient solution methods are important because practical optimization problems, even when simplified, can be very large.

In a linear optimization (LO) problem, a linear function is minimized or maximized, subject to a number of linear equalities and inequalities in which all the variables are continuous. Even though this is a restrictive class of problems, many applications of LO exist. One way to reduce the solution time for large LO problems is to exploit more powerful computer hardware and, in particular, parallel computers, which provide a relatively inexpensive source of computational power. In this article we describe the parallel capabilities recently added to our LO software, called MOSEK.

APPLICATIONS ON ADVANCED ARCHITECTURE COMPUTERS

Greg Astfalk, Editor

A Parallel Interior-point Optimizer

The core of MOSEK consists of several algorithms for solving the LO problem:

$$\begin{aligned} &\text{minimize} && c^T x \\ &\text{subject to} && Ax = b \\ & && x \geq 0 \end{aligned} \qquad (1)$$

where $c \in R^n$, $A \in R^{m \times n}$, and $b \in R^m$ are parameters and $x \in R^n$ are the decision variables. In practice, m and n can range from one to several million. For most problems, the matrix A is almost always sparse; for typical large-scale problems, more than 99% of the entries of A are zero.

MOSEK offers two algorithms for the solution of (1): a primal simplex algorithm, the classical method for solving (1), and an interior-point algorithm. Recently, interior-point algorithms have become a strong competitor of the simplex method. Interior-point algorithms differ from the simplex algorithm in generating, rather than a sequence of extreme-point solutions, a sequence of points in the interior of the positive orthant. These points converge toward the optimal solution. Theoretically, interior-point algorithms have excellent convergence properties because of their polynomial complexity, first proved in [3]. This theoretical efficiency is confirmed in practice. Almost independent of the size of (1), an interior-point algorithm solves the problem in 10–100 iterations.

A single iteration of an interior-point algorithm, which involves a Cholesky factorization of a matrix of dimension $m \times m$, is much more computationally expensive than an iteration of the simplex algorithm. The computational complexity of one simplex iteration is $O(mn)$, whereas that of an interior-point iteration is $O(n^3)$.

Not much work is performed at each iteration of the simplex algorithm, and the advantage that can be gained from parallelization is therefore limited. Furthermore, the simplex iterations themselves must be performed sequentially.

For an interior-point algorithm, however, the computationally expensive iterations turn out to be an advantage because they can be parallelized. Therefore, our focus here is the parallelization of the interior-point algorithm.

We chose the Silicon Graphics shared-memory platform as our initial architectural target for several reasons. First, the shared-memory architecture eliminates most of the problems of data distribution. Second, MOSEK is coded in C, making the SGI platform, with its Power C language, a good development environment. Third, other platforms, like the parallel Suns and Hewlett–Packards, are similar to the SGI. At a later stage, it should be easy to port the SGI code to those platforms as well. A recently defined portable interface called OpenMP (see http://www.openmp.org/) provides facilities that are close to the extensions included in the Power C language on several hardware platforms. OpenMP is supported by Compaq (a.k.a. Digital), Hewlett–Packard, IBM, Intel, SGI, and Sun.

To make a program capable of exploiting multiple processors, it is necessary to insert compiler directives into the code in the form of “pragmas.” These language constructs are illustrated by the following code fragment:

```
#pragma parallel
        #pragma  pfor(i=0; 10000; i)
{
   for ( i=0 ; i<10000; ++i)
     x[i] = y[i];
}
```

In this example, Power C will divide the loop into a number of independent parts, corresponding to a range of indices. Each part is then executed by its own thread, and each thread is allocated to a processor. Hence, the loop is parallelized.

The pragma, with its instruction to the compiler that the loop can be parallelized, is needed because not all loops can safely be automatically parallelized. To improve parallel efficiency, it is necessary to reorganize the code in such a way that more loops can be parallelized. An important issue in the code reorganization is that it should be possible to divide the work of a loop equally among the processors such that good load balancing is achieved.

Due to space limitations, we cannot discuss the MOSEK interior-point algorithm in detail here; the interested reader is referred to [1, 2]. Because the interior-point algorithm in MOSEK was not initially designed for parallel execution, it was necessary to insert pragmas into the source code and reorganize some of the computations to give it parallel capabilities.

Each iteration of an interior-point algorithm involves the solution of a set of linear equations resulting from the application of Newton's method to a set of nonlinear equations. Solving these linear equations is nearly equivalent to solving the normal equations in the methods used to solve linear least-squares problems.

The two most computationally expensive operations in this approach are the matrix–matrix product

$$AA^T \qquad (2)$$

and the Cholesky decomposition

$$AA^T = LL^T \qquad (3)$$

where L is a positive-definite lower triangular matrix. Because the columns of A are rescaled in each iteration of the interior-point algorithm, the computations (2) and (3) are repeated in each iteration. After the Cholesky decomposition has been computed, the coefficient matrix L and L^T are used to solve several systems of linear equations. For certain LO problems, matrix–vector products of the form

$$Ax \qquad (4)$$

and

$$A^Ty \qquad (5)$$

can also be computationally expensive (x and y are known vectors of appropriate dimension).

The computation of (5) in parallel is easy because, in MOSEK, the A matrix is stored in a sparse column-wise format. We can thus parallelize the matrix–vector product (5) by letting each processor independently compute the inner products between y and a subset of the columns of A.

This approach cannot be applied to evaluate (4) in parallel when A is stored column-wise. If A is stored row-wise, however, an approach identical to that used for the evaluation of (5) can be used for (4). Whenever the MOSEK interior-point algorithm is executed in parallel, A is stored both row-wise and column-wise. At the cost of some additional storage, then, parallelization becomes trivial; in practice, owing to the sparsity of A, the storage penalty is not significant.

For most LO problems the computation of (4) and (5) is not time consuming, whereas the computation of (2) and the Cholesky decomposition (3) can require between 50% and 80% of the total computation time. Forming the matrix–matrix product can obviously be done in parallel, because two columns of the matrix (2) can be computed independently. We parallelize this operation by giving each processor responsibility for the computation of a subset of the columns in (2).

Parallel Cholesky

Once (2) has been computed, the Cholesky decomposition is performed. Doing the Cholesky decomposition in parallel is not a trivial task. One complication is that A is an unstructured sparse matrix, which implies that AA^T and, hence, L are sparse. Exploitation of the sparsity is important for reducing both the storage requirements and the amount of computation.

It is well known that the sparsity of L is dependent on the ordering of the rows and columns of AA^T. Therefore, we would like to perform a symmetric ordering of the rows and columns of (2) such that the number of non-zeros in L is minimized. This is not possible: Choosing an optimal ordering is an NP-complete problem. It is thus only heuristics, such as the minimum-degree algorithm, that can be used. In MOSEK the first step is to compute a symmetric ordering of the matrix (2), after which the sparse data structure for L is created. This ordering is a one-time effort.

Without loss of generality, we assume that A is of full row rank; the matrix (2) is then positive-definite and symmetric. If L is initialized with (2), it is easy to verify that

$$\begin{bmatrix} L_{11}L_{21} \\ L_{21}L_{22} \end{bmatrix} = \begin{bmatrix} L_{11}^{\frac{1}{2}} & 0 \\ L_{21}L_{11}^{-\frac{1}{2}} & I \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & L_{22} - L_{21}L_{11}^{-\frac{1}{2}}(L_{21}L_{11}^{-\frac{1}{2}})^T \end{bmatrix} \begin{bmatrix} (L_{11}^{\frac{1}{2}})^T (L_{21}L_{11}^{-\frac{1}{2}})^T \\ 0 \quad I \end{bmatrix} \qquad (6)$$

Here L_{11} is square and $L_{11}^{-1/2}$ is the Cholesky decomposition of L_{11}. Since L is positive-definite, the matrix

$$L_{22} - L_{21}L_{11}^{-\frac{1}{2}}(L_{21}L_{11}^{-\frac{1}{2}})^T \qquad (7)$$

is also positive-definite. The decomposition (6) can be applied recursively, and an algorithm for computing the Cholesky decomposition is obtained. In particular, if L_{11} is chosen to be a 1×1 block, the algorithm is very simple. It should be emphasized that the resulting algorithm is mathematically equivalent to Gaussian elimination on L, with the pivot element chosen to be on the diagonal.

The update (7) can be performed immediately after

$$L_{21}L_{11}^{-\frac{1}{2}} \qquad (8)$$

has been computed. The update of a particular part of (7) can also be delayed until this part is needed in the computations. If the update is performed immediately, the resulting algorithm is called a “push Cholesky”; otherwise, it is called a “pull Cholesky.”

If L is a fully dense matrix, push Cholesky can easily be parallelized, as follows: First, k is chosen to be a small multiple of the number of available processors; L_{11} is the leading $k \times k$ block of L. $L_{11}^{-1/2}$ is then computed sequentially on a single processor. This is followed by the computation of (8) and (7). These two tasks can easily be split into small independent tasks, which can be accomplished on several processors. Since the amount of work performed in each subtask is known, good load balancing can be obtained. Hence, if L is a dense, or nearly dense, matrix, we know how to compute L efficiently in parallel. As mentioned earlier, however, L tends to be a large sparse matrix.

For instance, L can be of the form

$$L = \begin{bmatrix} x & & & & & & & & \\ & x & & & & & & & \\ x & & x & & & & & & \\ & & & x & & & & & \\ & x & & & x & & & & \\ & & & & & x & & & \\ & & x & & & x & x & & \\ x & & x & & & x & x & x & \\ & & & x & x & & & x & x \\ & x & & x & x & & & x & x & x \end{bmatrix} \qquad (9)$$

Each x in (9) denotes a non-zero; all the remaining coefficients are zero. The upper part of L can be seen to be very sparse, whereas

the lower triangular corner of the matrix forms a more dense submatrix. This observation holds ingeneral for the Cholesky decompositions arising in the interior-point algorithm implemented in MOSEK, because of the way the symmetric reordering of AA^T is done.

The sparsity in L actually turns out to be an advantage, because it implies that some of the computations can be performed independently. In the example L matrix, column 1 updates column 3 and column 2 updates column 5. In this way the sparsity of L gives rise to these tasks, which can be performed independently. This is of obvious advantage for parallelization.

Elimination Trees

To describe the dependencies in the elimination process, we introduce the elimination tree [4], which is defined as:

$$parent[j] := \begin{cases} 0, & l_{kj} = 0, \quad \forall k > j \\ \min\left\{k \middle| l_{kj} \neq 0\right\}, & \text{else} \end{cases}$$

Additionally,

$$T[j] := \text{the subtree rooted at node } j \text{ in the elimination tree}$$

and

$$H[T[j]] := \{\text{height of the tree}\}\ T[j]$$

Thus, all the subtrees rooted at the leaf nodes have height 1, all the subtrees rooted at the parents of the leaf nodes have height 2, and so forth. Finally, we define

$$sublevel[k] := \{j: H[T[j]] = k\}$$

implying that all subtrees that have the same height belong to the same sublevel. For the example (9), the *parent* and *sublevel* are shown in Table 1.

node(j)	1	2	3	4	5	6	7	8	9	10
parent[j]	3	5	7	9	9	7	8	9	10	0
$H[T[j]]$	1	1	2	1	2	1	3	4	5	6

Table 1. *Definition of* parent *and* sublevel *for the example (9).*

Even though this algorithm runs in parallel, it has some disadvantages. Unless the number of nodes at each sublevel is larger than the number of processors, and the amounts of work required to update all the nodes on a sublevel are fairly similar, use of this algorithm does not lead to good load balancing. Stated differently, the algorithm requires that the elimination tree be well balanced. Unfortunately, it is well known that the minimum-degree ordering does not generate a well-balanced elimination tree.

From the construction of the elimination tree, it follows that the number of nodes on each sublevel decreases as the sublevel index increases. The size of the subtrees increases as well. If cut at a certain level, denoted by *cutlevel*, the elimination tree splits into a large number of small subtrees and one fairly large parent tree. This can be illustrated by the example

$$\begin{bmatrix} L_{11} & & & & & \\ & L_{22} & & & & \\ & & \ddots & & & \\ & & & L_{kk} & & \\ L_{(k+1)1} & L_{(k+1)2} & \cdots & L_{(k+1)k} & L_{(k+1)(k+1)} \end{bmatrix} \tag{10}$$

in which it is assumed that k subtrees appear when the elimination tree is cut off at a given level. It is further assumed that all the nodes corresponding to one subtree are ordered sequentially, which means that block L_{ii} corresponds to all the nodes of one subtree. The last block, $L_{(k+1)(k+1)}$, is the parent tree, which can be assumed to be fairly dense, given an appropriate choice for the cutlevel.

Clearly, the small subtrees corresponding to the first k blocks can be processed in parallel. Moreover, if the cutlevel is chosen appropriately, the number of subtrees available for distribution among processors will be large enough to achieve good load balancing. For factorization of the last (k+1) block, the following procedure is used to factorize all the nodes, one sublevel at a time: First, all the nodes at the current sublevel are updated with the information from the first k blocks. To distribute the work equally among processors, a list for each node on the current sublevel is formed. The list contains the nodes from the first part that are used to update the block (notice that any node in the first part can appear in only one list). All the nodes at the current sublevel, considered as one matrix, are then updated in parallel by techniques similar to those used for a parallel push-Cholesky update. When all the nodes on a given sublevel have been updated, each node is factored with a parallel dense Cholesky decomposition (with all processors used to factor one node at a time).

	Preprocessed		Non-zeros	
Name	Rows	Columns	$\lvert AA^T \rvert$	$\lvert L \rvert$
bitesest	30643	271596	625601	4000913
chinese	8067	41154	226851	68408
dbir1	7154	24862	1112931	2279495
dfl001	3810	8910	40103	798902
ken-18	39856	89347	177502	1274821
many	20145	183062	482605	3510790
mod2	21769	24351	905326	905326
nug15	6630	22275	115204	4538918
pds-20	6834	71354	140995	1238728
stormG2-125	47786	129994	286161	589661

Table 2. *The test problems.*

		Solution time (s)			Speedup	
Platform		1	2	4	2	4
SGI	**bitest**	794	476	460	1.7	1.7
	chinese	185	106	88	1.7	2.1
	dbir1	332	184	154	1.8	2.2
	dfl001	81	55	35	1.5	2.3
	ken-18	128	90	71	1.4	1.8
	many	1272	759	556	1.7	2.3
	mod2	128	94	78	1.4	1.6
	nug15	1243	712	509	1.7	2.4
	pds-20	246	166	132	1.5	1.9
	stormG2-125	403	253	221	1.6	1.8
PC	**bitest**	1445	746		1.9	
	chinese	229	172		1.3	
	dbir1	490	315		1.6	
	dfl001	138	92		1.5	
	ken-18	155	132		1.2	
	nug15	2075	1321		1.6	
	many	1931	1398		1.4	
	mod2	183	148		1.3	
	pds-20	347	237		1.5	
	stormG2-125	804	568		1.4	

Table 3. *Timing results.*

Computational Results

At this point, we have discussed the major tasks that have been parallelized within the MOSEK interior-point algorithm. Certain other less time consuming tasks have also been parallelized but are not discussed here.

We emphasize that our implementation is not intended to be scalable for a large number of processors. Nevertheless, it is our experience that if the computation of the Cholesky decomposition is the dominant computational cost in the solution of (1), our implementation does scale reasonably well on systems of up to 16 processors.

To show the effectiveness of our approach, we present computational results obtained with MOSEK. Table 2 gives the statistics for our test problems, which were chosen to represent different problem domains, such as multi-commodity network flows, stochastic programming, and LP relaxations of quadratic assignment and set-covering problems.

The table shows the numbers of rows (m), columns (n), and non-zeros in AA^T and L. The dimension of L is identical to the number of rows, and the matrix L for **nug15** is therefore dense. Before optimization of the problems with MOSEK, the software package XPRESS-MP was used to preprocess and rescale them.

Table 3 presents computational results for the test problems. The problems were run on a 16-processor SGI Origin 2000 computer with 2 gigabytes of memory and a two-processor PC computer that has 128 megabytes of memory and runs Windows NT.

Table 3 shows the solution times (in CPU seconds) for the test problems run on the master processor with one, two, and four processors. The speedups are also shown. (The definition of speed-up on k processors is the time required to solve the problem with one processor divided by the time required with k processors.) In general, ideal linear speedups are not obtained, in part because not all operations are parallelized and in part because of the load-balancing effects. This is especially true for computationally inexpensive problems like **ken-18**. For computationally expensive problems, however, good speedups are achieved; even on the

inexpensive PC hardware, a respectable speedup is realized on two processors.

Conclusion

Even though the work of an interior-point algorithm cannot be split into a number of independent tasks, we have obtained good speedups for compu-tationally expensive problems on shared-memory machines. The speedup is realized on both the SGI platform and the less expensive PC platform. The implementation described here is not scalable to a large number of processors; PC hardware, though, is not likely to be equipped with large numbers of processors any time soon.

The Web site for the MOSEK software discussed in this article is http://www.mosek.com.

References

[1] E.D. Andersen and K.D. Andersen, *A parallel interior-point based linear programming solver for shared-memory multiprocessor computers: A case study based on the XPRESS LP solver*, Technical report, CORE, UCL, Belgium, 1997.

[2] E.D. Andersen and K.D. Andersen, *The APOS linear programming solver: An implementation of the homogeneous algorithm*, Technical Report Publications from Department of Management No. 2/1997, also available as CORE Discussion Paper 9730, Department of Management, Odense University, Denmark, 1997.

[3] N. Karmarkar, *A polynomial-time algorithm for linear programming*, Com-binatorica, 4 (1984), 373–395.

[4] J.W.H. Liu, *The role of the elimination tree in sparse factorization*, SIAM J. Mat. Anal. Appl., 11:1 (1990), 134–172.

Erling Andersen (e.d.andersen@twi.tudelft.nl) is a TMR Research Fellow at TU Delft, The Netherlands. Knud Andersen (kda@eka.globalnet.co.uk) is a software developer at Dash Associates in Royal Leamington Spa, UK.