CENTER FOR
PARALLEL OPTIMIZATION

MASSIVELY PARALLEL SOLUTION OF QUADRATIC PROGRAMS VIA SUCCESSIVE OVERRELAXATION

by

R. De Leone and M. A. Tork Roth

Computer Sciences Technical Report #1041

August 1991

MASSIVELY PARALLEL SOLUTION OF QUADRATIC PROGRAMS VIA SUCCESSIVE OVERRELAXATION *

R. DE LEONE† AND M. A. TORK ROTH†

Abstract. Serial and parallel successive overrelaxation (SOR) solutions of specially structured large scale quadratic programs with simple bounds are discussed. By taking advantage of the sparsity structure of the problem, the SOR algorithm was successfully implemented on two massively parallel Single–Instruction–Multiple–Data machines: a Connection Machine CM–2 and a MasPar MP–1. Computational results for the well–known obstacle problems show the effectiveness of the algorithm. Problems with millions of variables have been solved in a few minutes on these massively parallel machines, and speedups of 90% or more were achieved.

Key words. Quadratic Programs, Successive Overrelaxation, Massively Parallel Algorithms.

1. Introduction.

Our concern here is the quadratic programming problem with simple bounds:

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}x^T Mx + q^T x \\ \text{subject to} & l \le x \le u \end{array} \tag{1.1}$$

where M is a symmetric, positive semidefinite nxn real matrix with positive diagonal entries and q, l, u, and x are n–dimensional vectors. Problems of this form arise in many physical and engineering applications such as contact and friction problems in rigid body mechanics, elastic torsion problems, and journal bearing lubrication [1, 2]. Least squares problems with box constraints are also a special case of (1.1), and sequential quadratic programming algorithms for nonlinear optimization with bound constraints require the solution of a quadratic problem such as (1.1) at each iteration [8, 9].

In recent years a large amount of research has been devoted to the solution of large scale quadratic programming problems. Two classes of algorithms have received particular attention: active set strategies and interior point methods.

Active set strategies which allow adding or dropping only one constraint from the set of active constraints at each iteration are clearly unsuitable for extremely large problems (with millions of constraints). Algorithms that are able to drop and add many constraints at each iteration were proposed by Dembo and Tulowitzki [6], Yang and Tolle [21], Wright[20], Moré and Toraldo [19] and Moré [18]. These algorithms use a gradient projection method until a suitable set of active constraints is identified. The face of the feasible region defined by the current iterate is then explored via a conjugate gradient method. In particular, the effective GPCG algorithm proposed by Moré and Toraldo [19] uses a gradient projection method until a suitable set of active constraints is identified or the gradient projection method fails to make reasonable progress. In this

* This material is based on research supported by the Air Force Office of Scientific Research Grant AFOSR-89-0410

† Center for Parallel Optimization, Computer Sciences Department, University of Wisconsin Madison, 1210 West Dayton Street, Madison, WI 53706

case a conjugate gradient method is used in conjunction with a projected line search strategy. Convergence in a finite number of iterations is established under reasonable conditions on the matrix M. The main disadvantage of this class of algorithms is the substantial effort devoted to exploring faces that do not contain an optimal solution. In addition, the performance of the algorithm depends on the strategy used to determine whether the projected gradient or the conjugate gradient is to be used.

More recently, polynomial time algorithms for quadratic programming problems with box constraints have been proposed [22, 17, 10]. (See [10] for a brief outline of the major results with interior point methods for quadratic programs). The algorithm by Han, Pardalos and Ye [10] requires a total of $O(\sqrt{n}L)$ iterations, where L is the size of the input data of the problem. To our knowledge, this is the best theoretical result to date. However, at each iteration, $O(n^3)$ arithmetic operations are required since a linear system of equations must be solved. This is the major time–consuming step of the algorithm.

The aim of this paper is to show that serial and parallel successive overrelaxation (SOR) methods [12, 13] can efficiently solve extremely large sparse quadratic programs. Although successive overrelaxation is intrinsically a serial algorithm, synchronous and asynchronous parallel versions were proposed [14, 15, 3, 4, 5]. The effectiveness of these algorithms in solving large sparse linear complementarity problems and linear programs depends on the fact that the algorithms are sparsity–preserving and the original data are never modified. We present here new computational results for specially structured quadratic programs. Taking advantage of the sparsity structure of the matrix M, we implemented the SOR algorithm on massively parallel Single–Instruction–Multiple–Data (SIMD) machines. Problems with over one million variables were solved to a high degree of accuracy in less than 5 minutes. A comparison of solution times with the Han, Pardalos and Ye algorithm is given in Section 4, and they are on the average nearly 6.5 times faster.

The paper is organized as follows. In Section 2 we briefly describe the serial SOR algorithm and present some implementation details that substantially decrease computing time. In Section 3 we discuss issues concerning two parallel implementations (in Fortran–90 and C*) that have been carried out on two different massively parallel machines: the MasPar MP–1 and the Connection Machine CM–2. Finally, in Section 4 we give results for both the serial and the parallel versions for the well–known obstacle problems.

We briefly describe our notation now. Given the vectors l and u (with $l \leq u$) and a vector x all in $\mathbb{R}^n$, $x_\#$ will denote the vector with components $(x_\#)_i = \min\{u_i, \max\{l_i, x_i\}\}$. The scalar product of two vectors x and y in $\mathbb{R}^n$ will be denoted by $x^T y$. For A in $\mathbb{R}^{m x n}$, A_i will denote the ith row of A. The symbol $:=$ denotes definition of the term on the left side of the symbol.

2. The serial SOR algorithm. In this section we will discuss a serial implementation of the SOR algorithm for the quadratic program with simple bounds (1.1).

It is easy to verify directly (or see [12]) that x is a solution of (1.1) if and only if

$$x = (x - \omega E(Mx + q))_{\#}$$

for some $\omega > 0$ and for some positive diagonal matrix E. This simple relationship is the basis of the algorithm. The successive overrelaxation algorithm constructs a sequence of iterates $\{x^k\}$ as follows:

SOR Algorithm

For any initial feasible $x^0 \in [l, u]$, generate the sequence $\{x^k\}$, $k = 1, 2, \ldots$, as follows:

$$x_i^{k+1} := \left(x_i^k - \omega E_{ii} \left(\sum_{j<i} M_{ij} x_j^{k+1} + \sum_{j \geq i} M_{ij} x_j^k + q_i \right) \right)_{\#} \tag{2.1}$$

for $i = 1, 2, \ldots, n$.

In our implementation we used $E_{ii} = M_{ii}^{-1}$. Convergence of the algorithm can be established if the relaxation parameter ω is chosen in the interval $(0, 2)$ [11].

It should be noted that two distinct components x_i and x_j (with $i < j$) can be concurrently updated provided that $M_{jr} = 0$ for all $r = i, i+1, \ldots, j-1$. This observation is the basis of our massively parallel implementation of the algorithm for specially structured matrices. Moreover, the components of x need not be updated in the natural order; a permutation of the indices $\{1, 2, \ldots, n\}$ can be defined and the components of x updated accordingly.

At iteration k, to update the ith component of the solution vector x, the scalar product of the ith row of M and a vector whose components are x_j^{k+1} for $j < i$ and x_j^k for $j \geq i$ must be computed. Since this is the most time–consuming part of the algorithm, special care must be taken to define a data structure for the matrix M that facilitates quick retrieval of all elements of a particular row. In our serial implementation, we assumed that the matrix M is sparse. However, no special sparsity structure was assumed a priori. We therefore stored the matrix M as a collection of sparse vectors, each vector being a row of M. Two integer arrays JA and IA and one real array AS were required for this scheme [7].

The choice of the relaxation parameter ω was crucial for the performance of the algorithm. We experimented with different fixed values of ω as well as an updating scheme for the relaxation parameter. In the latter case, the value of ω was monotonically decreased to 1.0 as the algorithm converged toward the solution. The best results (for the class of problems considered here) were obtained for very high values of ω between 1.8 and 1.95. When a value of ω less than 1.0 was used, the number of iterations increased dramatically.

Another simple but very effective procedure reduced total solution time. During the updating process, the algorithm recorded if (and how many times) a particular variable remained fixed at its upper or lower bound. A flag was associated with each component

of the solution vector and only components with value of the flag equal to 0 were updated using (2.1). If the variable remained fixed at its bound for a certain number of iterations, the flag was set to 1. From time to time, it was necessary to check if the decision to fix a particular variable was still correct. This was achieved by setting the flags for all the variables back to 0 and executing a new iteration of the SOR algorithm. If the value of a variable that was previously fixed did not change during this new iteration, the flag for this variable was set to 1 again. In addition, the algorithm checked for new variables to fix. In our implementation, this resetting step was performed every 30 iterations. For the test problems we considered, the successive overrelaxation algorithm was able to quickly identify (after 50 iterations or less) almost all the variables that would be either at the lower or at the upper bound at the optimum. A 15% reduction in solution time was achieved by implementing this bound–fixing strategy.

3. The massively parallel SOR algorithm. The quadratic programming problems considered in our parallel implementations arise as a finite element approximation to elliptical variational inequalities. This approximation is obtained by triangulating the unit square, giving rise to a grid. The grid points represent the values of a piecewise linear function at the vertices of the triangulation. For a more complete description of the problems see [19]. We experimented with a Fortran–90 implementation on both a MP–1 and CM–2, as well as a C* implementation on the CM–2. The two most important issues in the parallel implementations were minimizing communication between processors, and ensuring that all processors were busy doing useful work at any given time step.

The special pentadiagonal structure of the matrix M allowed us to take advantage of the NEWS (North–East–West–South) communication grid of the CM–2, and X–Net communication on the MasPar MP–1. In this context, it is more convenient to describe the SOR algorithm and our implementation not in terms of an n–dimensional solution vector x, but rather in terms of an $m \times m$ solution matrix X, where $n = m^2$. The components of X correspond to the values at the points in the grid. To update a particular component X_{ij}, the SOR algorithm requires the value X_{ij}, along with its north, south, east and west neighbors in X.

By imposing a standard red–black coloring [16] upon the grid to determine the components of X that can be updated concurrently, individual components were assigned to processors in such a way that each processor would be active at every time step. All red components were updated simultaneously, followed by all black components.

In our C* implementation, we were able to take advantage of the *shape* construct in order to define the shape of the machine to fit the natural description of the data. The processors were organized in an $p \times p$ grid, with each processor being assigned a $k \times k$ submatrix of X, and $n = p^2 \times k^2$. In this way, communication between processors was reduced to sharing data values along the common interior borders of X. This was achieved by the C* library function *pcoord*. The values needed to update a component interior to the $k \times k$ submatrix were all local (that is, they were in the local memory of the processor to which that component had been assigned). To update boundary components of the submatrix, a processor had to access at most two components from

neighboring processors in the NEWS grid.

We implemented a Fortran–90 version of the code on both the MasPar MP–1 and the CM–2. Data layout directives were not yet available with the MasPar Fortran–90 compiler version we used. On the MasPar MP–1, parallel data are allocated across a two–dimensional processor grid (for instance, processor $(1,1)$ in the grid would be assigned element $(1,1)$ of a two–dimensional array, and so on). In our implementation, the components of X were stored in four interleaving $m/2 \times m/2$ submatrices. With this arrangement, each processor was assigned four components of X; two to be updated in a red iteration and two to be updated in a black iteration. Furthermore, two of the four values of X needed to update a particular component were in a processor's local memory. The remaining two were in the local memory of neighboring processors, and were accessed using the Fortran–90 construct EOSHIFT.

4. Performance of the serial and parallel algorithms. In this section we present computational results for both serial and the parallel versions of our SOR algorithm.

The test problems considered are instances of the well–known obstacle problem which consists of finding the equilibrium position of an elastic membrane subject to a vertical force. This problem can be posed as the following constrained variational problem:

$$\min\{q(v) : v \in K\}$$

with

$$q(v) = 1/2 \int_{\mathcal{D}} \|\nabla v\|^2 \, d\mathcal{D} - f \int_{\mathcal{D}} v d\mathcal{D},$$

$$\mathcal{D} = (0,1) \times (0,1),$$

∇ is the Laplacian operator, and K is the subset of all functions v with compact support on $\mathcal{D}$ such that v and $\|\nabla v\|^2$ belong to the square integrable class $L^2(\mathcal{D})$, and v varies between obstacle bounds v_l and v_u. For our experiments, the membrane force $f = 1$. Finite element approximations give rise to a minimization of a quadratic problem with a finite number of variables. The matrix M is the pentadiagonal matrix with diagonal entries of 4 and off–diagonal entries of -1 that arises from a finite difference approximation to the Laplacian operator. In all problems considered here, the matrix M is identical. The lower and upper obstacle bounds for each of the six problems are given in Table 1.

In Tables 2 to 5 we report computational results for the serial version of the SOR algorithm. The results were obtained on an IBM RISC 6000 POWERstation 550. All numerical results were obtained in double precision.

Table 2 shows the number of iterations, solution times (in seconds) and number of active constraints at the optimum, i.e. the number of variables at the lower or upper bound. The number of variables varies from 10,000 to 490,000 which corresponds to a

Problem	v_l	v_u
1	$(\sin(9.2x_1)\sin(9.3x_2))^3$	$(\sin(9.2x_1)\sin(9.3x_2))^2 + 0.02$
2	$\sin(3.2x_1)\sin(3.3x_2)$	2000.0
3	$0.3(\sin(3.2x_1)\sin(3.3x_2)$	2000.0
4	$(\sin(3.2x_1)\sin(3.3x_2)^2$	2000.0
5	$(\sin(3.2x_1)\sin(3.3x_2)^3$	2000.0
6	$(16x_1(1-x_1)x_2(1-x_2))^3$	$(16x_1(1-x_1)x_2(1-x_2))^2 + 0.01$

TABLE 1: *Lower and upper obstacle bounds*

grid with the number of points varying from 100x100 to 700x700. The accuracy of the solution defined as

$$\text{Accuracy} := \left\| x - (x - (Mx + q))_{\#} \right\|_{\infty}$$

is reported in the fifth column while the column labeled HPY reports the solution time for the same problems with the Han–Pardalos–Ye interior point algorithm [10] on a IBM 3090–600S supercomputer with Vector Facilities. Moreover, in their algorithm, the conjugate gradient method used for solving the system of linear equations took advantage of the special pentadiagonal structure of the matrix M. As we stated earlier, in our serial implementation we took advantage of the sparsity structure of the matrix, but special sparsity was not assumed a priori. For all the problems we tested, the solution time for our serial SOR algorithm was lower than the time required by the Han–Pardalos–Ye algorithm. The solution time ratio for the two algorithms is shown in the last column. The time per iteration for the SOR algorithm grows linearly with the number of variables while the number of iterations grows sublinearly. Finally, we note that for this problem, about 1/5 of the variables are at their lower or upper bound at the optimum. The algorithm was able to identify a large portion of these variables quickly. By applying the fixing strategy described in Section 2, a 15% reduction in solution time was achieved.

Table 3 reports the computational results for Problem 2. For this problem, almost 60% of the variables are fixed at the optimum. In this case, the reduction achieved by implementing the fixing strategy was even greater.

In next two tables we report number of iterations, solution time in seconds and number of variables at the lower or upper bounds at the optimum for the remaining 4 problems (Problems 3–6). A larger number of iterations were required for Problems 3, 4 and 5. In addition, a small percentage of the variables are at the lower or upper

bound for Problem 5. This explains the greater solution time required for this problem. All problems were solved with an accuracy of 10^{-7} or better.

Figure 1 shows the pattern of active constraints for the six problems considered in this paper. Shaded areas depict either upper or lower bound active constraints.

Tables 6–9 report the solution times for the massively parallel Fortran–90 and C* implementations of the SOR algorithm on both the Connection Machine CM–2 and the MasPar MP–1.

A word of caution is in order now in interpreting SIMD machine times. Solution time on the CM–2 is strongly influenced by exclusive/non–exclusive access to the partition and by the load on the front end. If access to the partition is not exclusive or if the load on the front end is very high, the solution time can vary by a large amount. In some cases we observed a variation by a factor of 2 or 3. Therefore, we attempted to execute our program with exclusive access to the 8K or 16K processor partition. In addition, we were careful to run our SOR algorithm only when the load on the front end was sufficiently low. In this sense the solution times reported here represent solution time in an "ideal" or "near–ideal" situation. However, despite all our efforts, in some cases we were unable to achieve these ideal conditions.

We make the following observations. First, since in our C* implementation communication between processors was reduced to sharing data along the common interior border, the solution time was lower than the time required for the Fortran–90 implementation. A 30–fold improvement over the serial algorithm was achieved for sufficiently large problems. By doubling the number of processors from 8K to 16K, we were able to cut the total solution time by almost 50%. The speedup was well over 90% for sufficiently large problems. Finally, the number of iterations required increased almost linearly with the number of variables.

Table 10 reports solution times for an MPL (MasPar Parallel Application Language) implementation on a Maspar MP–1 with both 8K and 16K processors. The code was written and the results obtained by Mark Holt of MasPar Computer Corporation. The code uses the X–Net communication protocol to share information along the common interior borders. The function Profiler available under MPPE (MasPar Programming Environment) showed that the amount of time spent communicating substantially decreased as the size of the problem increased. In fact, about 38% of the total time was spent sharing information between processors for $n = 65,536$. When $n = 1,048,576$, this accounted for less than 10% of the total execution time.

5. Conclusions. We have presented computational results for a serial and a massively parallel implementation of the SOR algorithm for large–scale quadratic programs. Problems with more than a million variables were solved in less than 5 minutes on a 16,384-processor CM–2 machine. Moreover, a problem with almost 9.5 million variables was solved on the MP–1. To the best of our knowledge, this is the largest problem of this class that has been solved thus far.

6. Acknowledgments. We would like to thank Thinking Machines Corporation for providing us access to the CM–2 and MasPar Computer Corporation for providing

us access to the MP–1. In addition, we are grateful to Jill Mesirov of Thinking Machines for her suggestions in the early stages of development of the massively parallel implementation of the SOR algorithm, and to Mark Holt of MasPar for his advice and assistance in improving the design of the algorithm. In particular, it was his suggestion to implement a C version of the algorithm that led to significant performance gains.

REFERENCES

[1] G. Cimatti, *On a problem of the theory of lubrication governed by a variational inequality*, Applications of Mathematical Optimization, 3 (1977), pp. 227–242.

[2] G. Cimatti and O. Menchi, *On the numerical solution of a variational inequality connected with the hydrodynamic lubrication of a complete journal bearing*, Calcolo, 15 (1978), pp. 249–258.

[3] R. De Leone and O.L. Mangasarian, *Serial and parallel solution of large scale linear programs by augmented Lagrangian successive overrelaxation*, in Optimization, parallel processing and applications, A. Kurzhanski, K. Neumann, and D. Pallaschke, eds., vol. 304 of Lecture Notes in Economics and Mathematical Systems, Springer Verlag, Berlin, 1988, pp. 103–124.

[4] ——, *Asynchronous parallel succesive overrelaxation for the symmetric linear complementarity problem*, Mathematical Programming , Series B, 42 (1988), pp. 347–361.

[5] R. De Leone, O.L. Mangasarian, and T-.H. Shiau, *Multi-sweeep asynchronous parallel succesive overrelaxation for the nonsymmetric linear complementarity problem*, Annals of Operation Research, 22 (1990), pp. 43–54.

[6] R.S. Dembo and U. Tulowitzki, *On the minimization of quadratic functions subject to box constraints.* Working paper Series B #71, School of Organization and Management, Yale University, New Haven, 1983.

[7] I.S. Duff, A.M. Erisman, and J.K. Reid, *Direct methods for sparse matrices*, Oxford University Press, Oxford, England, 1989.

[8] U.M. Garcia Palomares and O.L. Mangasarian, *Superlinearly convergent quasi–Newton algorithms for nonlinearly constrained optimization problems*, Mathematical Programming, 11 (1976), pp. 1–13.

[9] W. Gill, P.E. Murray and M.H. Wright, *Practical Optimization*, Academic Press, 1981.

[10] C. Han, P.M. Pardalos, and Y. Ye, *Solving some engineering problems using an interior-point algorithm*, Tech. Report CS-91-04, Department of Computer Science, The Pennsylvania State University, Pennsylvania, 1991.

[11] Z.-Q. Luo and P. Tseng, *On the convergence of a matrix splitting algorithm for the symmetric monotone linear complementarity problem*, Tech. Report LIDS-P-1884, Laboratory for Information and Decision System, Massachusetts Institute of Technology, Cambridge, 1990. to appear *SIAM Journal on Control and Optimization.*

[12] O.L. Mangasarian, *Solution of symmetric linear complementarity problems by iterative methods*, Journal of Optimization Theory and Applications, 22 (1977), pp. 465–485.

[13] ——, *Sparsity-preserving sor algorithms for separable quadratic and linear programming*, Computer and Operation Research, 11 (1984), pp. 105–112.

[14] O.L. Mangasarian and R. De Leone, *Parallel successive overrelaxation methods for symmetric linear complementarity problems and linear programs*, Journal of Optimization Theory and Applications, 54 (1987), pp. 437–446.

[15] ——, *Parallel gradient projection successive overrelaxation for symmetric linear complementarity problems and linear programs*, Annals of Operation Research, 14 (1988), pp. 41–59.

[16] J.J. MODI, *Parallel Algorithms and Matrix Computation*, Clarendon Press, Oxford, England, 1988.
[17] R.D.C. MONTEIRO AND I. ADLER, *Interior path following primal-dual algorithms part II: convex quadratic programming*, Mathematical Programming, 44 (1989), pp. 43–66.
[18] J.J. MORÈ, *On the performance of algorithms for large-scale bound constrained problems*, Tech. Report MCS-P140-0290, Argonne National Laboratory, Argonne, Illinois, 1990.
[19] J.J. MORÈ AND G. TORALDO, *On the solution of large quadratic programming problems with bound constraints*, SIAM Journal on Optimization, 1 (1991), pp. 93–113.
[20] S.J. WRIGHT, *Implementing proximal point methods for linear programming*, Tech. Report MCS-P45-0189, Argonne National Laboratory, Argonne, Illinois, 1989.
[21] E.K. YANG AND J.W. TOLLE, *A class of methods for solving large convex quadratic programs subject to box constraints.* preprint, University of North Carolina, Department of Operation Research, Chapel Hill, North Carolina, 1988.
[22] Y. YE AND E. TSE, *An extension of Karmarkar's projective algorithm for convex quadratic programming*, Mathematical Programming, 44 (1989), pp. 157–179.

# vars	# iter	# fixed vars	time	accuracy	HPY	time ratio
10,000	306	2,412	6.30	$0.1848 * 10^{-7}$	16.3	2.59
40,000	323	8,761	26.86	$0.1643 * 10^{-7}$	131.1	4.88
90,000	361	19,104	68.10	$0.2027 * 10^{-7}$	437.6	6.43
115,600	393	24,237	95.89	$0.1900 * 10^{-6}$	700.3	7.30
160,000	449	33,096	148.54	$0.6603 * 10^{-8}$	1035.8	6.97
250,000	608	51,241	314.58	$0.5813 * 10^{-7}$	2110.5	6.71
360,000	872	73,288	750.58	$0.1171 * 10^{-7}$	4090.3	5.45
490,000	851	99,264	848.79	$0.9143 * 10^{-7}$	8977.8	10.58

TABLE 2: *Comparison of serial SOR algorithm on an IBM RISC 6000 POWERstation 550 and the HPY algorithm. Problem 1*

# vars	# iter	# fixed vars	time	accuracy	HPY	time ratio
10,000	191	6,157	3.53	$0.2056 * 10^{-6}$	25.4	7.20
40,000	417	24,120	32.03	$0.1244 * 10^{-7}$	203.9	6.37
90,000	459	53,840	80.44	$0.1288 * 10^{-6}$	699.9	8.70
115,600	535	69,072	124.15	$0.7085 * 10^{-7}$	1018.7	8.21
160,000	780	95,361	241.42	$0.6745 * 10^{-8}$	1534.7	6.36
250,000	1114	148,758	560.28	$0.5935 * 10^{-7}$	3141.9	5.61
360,000	1594	213,833	1357.48	$0.5659 * 10^{-7}$	5312.4	3.91

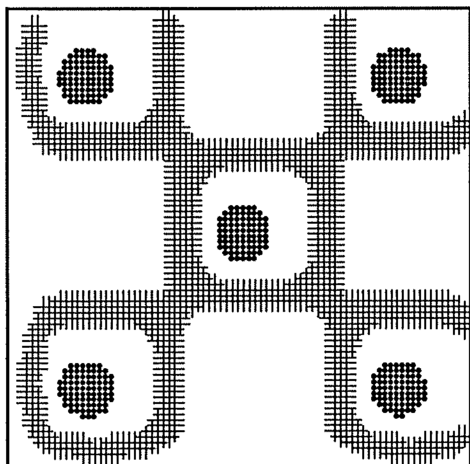
TABLE 3: *Comparison of serial SOR algorithm on an IBM RISC 6000 POWERstation 550 and the HPY algorithm. Problem 2*

# vars	Problem 3		Problem 4		Problem 5		Problem 6	
	# iter	time	# iter	time	# iter	time	# iter	time
10,000	309	5.63	336	7.49	329	7.55	264	5.21
40,000	473	37.69	430	38.70	410	38.35	317	24.73
90,000	794	153.78	831	170.39	1010	214.33	343	60.93
115,600	953	241.59	1076	283.91	1296	353.42	401	90.69
160,000	1304	453.45	1486	538.50	1777	663.12	519	157.83
250,000	1870	1052.96	1896	1093.18	2206	1308.79	588	281.79
360,000	2668	2560.67	2658	2639.07	3076	3112.20	782	627.63

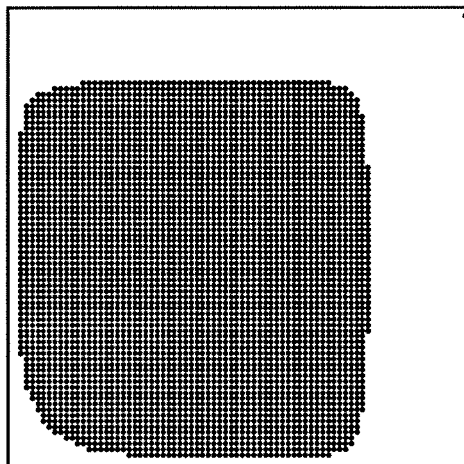
TABLE 4: *Solution time in seconds for the serial SOR algorithm on an IBM RISC 6000 POWERstation 550. Problems 3–6*

# vars	Problem 3	Problem 4	Problem 5	Problem 6
	# fixed vars	# fixed vars	# fixed vars	# fixed vars
10,000	4637	1321	704	3288
40,000	18192	5255	2725	12472
90,000	40658	11609	6071	27216
115,600	52129	14836	7772	34814
160,000	72026	20428	10727	47888
250,000	112208	31706	16687	74492
360,000	161240	45446	23963	106790

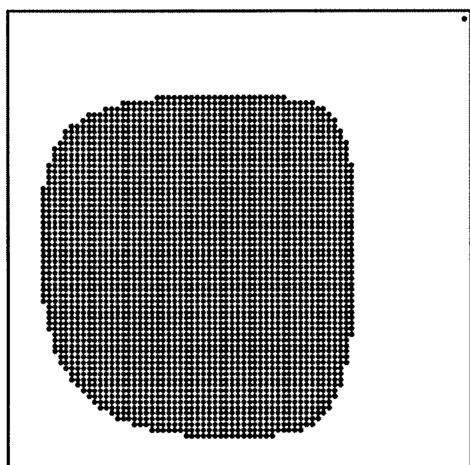
TABLE 5: *Number of variables fixed at the lower or upper bound at the optimum for serial SOR algorithm on an IBM RISC 6000 POWERstation 550. Problems 3–6*



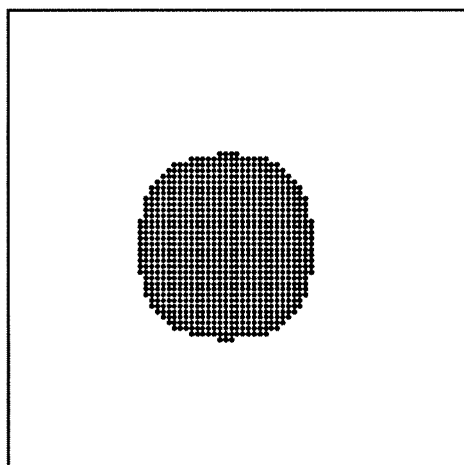
Problem 1



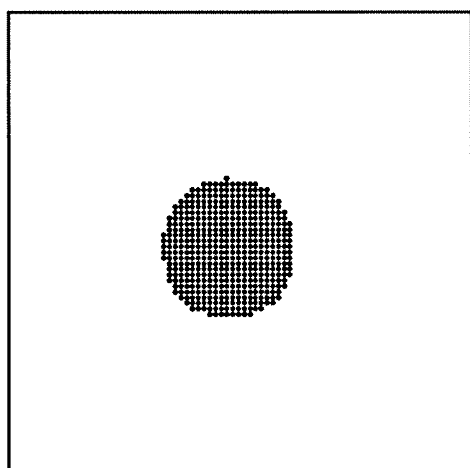
Problem 2



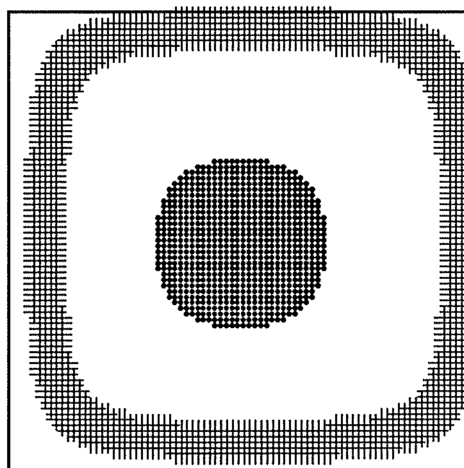
Problem 3



Problem 4



Problem 5



Problem 6

FIG. 1: *The pattern of active constraints for Problems 1–6: shaded areas depict active upper and lower bounds.*

n	# iter	CM–2	MP–1
90,000	480	8.3	7.8
160,000	620	18.6	13.4
250,000	980	57.4	20.6
360,000	1460	87.0	82.1
490,000	2000	139.7	150.6
640,000	2620	264.2	335.8
810,000	3330	546.9	496.0

TABLE 6: *Solution time in seconds for Fortran–90 implementation with 8K processors on the CM–2 and MP–1. Problem 2*

n	# iter	Fortran–90		C*	
		8K Procs	16K Procs	8K Procs	16K Procs
16,384	360	2.6	2.4	3.66	2.16
65,536	440	5.2	3.2	4.51	2.68
262,144	1040	57.2	34.8	32.04	18.57
1,048,576	4240		403.1	472.47	249.86

TABLE 7: *Solution time in seconds for Fortran–90 and C* implementations on the CM–2. Problem 2*

# vars	Problem 1		Problem 3		Problem 4		Problem 5		Problem 6	
	# iter	time	# iter	time	# iter	time	# iter	time	# iter	time
16,384	320	3.39	360	3.66	380	3.86	380	3.85	300	3.17
65,536	320	3.40	520	5.30	420	4.26	480	4.87	300	3.18
262,144	540	17.58	1740	53.83	1940	59.89	2300	70.92	340	11.06
1,048,576	2280	253.77	6900	767.85	7220	817.60	8420	1055.76	1320	147.79

TABLE 8: *Solution time in seconds for the parallel C* SOR algorithm on the CM–2 with 8K processors. Problems 1,3–6*

# vars	Problem 1		Problem 3		Problem 4		Problem 5		Problem 6	
	# iter	time	# iter	time	# iter	time	# iter	time	# iter	time
16,384	320	2.00	360	2.16	380	2.28	380	2.28	300	1.88
65,536	320	2.00	520	3.12	420	2.72	480	2.92	300	1.92
262,144	540	10.14	1740	31.12	1940	34.52	2300	40.91	340	6.42
1,048,576	2280	143.36	6900	406.67	7220	425.65	8420	541.17	1320	83.34

TABLE 9: *Solution time in seconds for the parallel C* SOR algorithm on the CM–2 with 16K processors. Problems 1,3–6*

n	# iter	8K Procs	16K Procs
65,536	440	2.3	1.3
262,144	1040	15.0	8.7
1,048,576	4240	216.0	109.5
4,194,304	15900	2750.0	1441.7
9,437,184	34200		6752.7

TABLE 10: *Solution time in seconds on the MasPar MP–1. Problem 2*