**TR/06/93** **August 1993**

# Solving Large Scale Linear Programming Problems using an Interior Point Method on Vector Processors

Ron Levkovitz

# Solving Large Scale Linear Programming Problems using an Interior Point Method on Vector Processors

Ron Levkovitz *

**Abstract**

The interior point method (IPM) is now well established as a computationaly competitive scheme for solving very large scale linear programming problems. The leading variant of the IPM is the primal dual predictor corrector algorithm due to Mehrotra. The main computational efforts in this algorithm are the repeated calculation and solution of a large sparse positive definite system of equations.

We describe an implementation of this algorithm for vector processors. At the heart of the implementation is a vectorized matrix multiplication and Cholesky factorization for sparse matrices.

We identify the parts where vectorization can be beneficial and discuss in details the merits of alternative vectorization techniques. We show that the best way to utilize a vector processor is by exploiting dense computation within the sparse framework and by unrolling loop operations. We further present an extended definition of supernodes, and describe an implementation based on this new approach. We show that although this approach requires more memory it can increase the scope of dense computation substantially with out adding extra operations.

Performance results on standard industrial test problems and comparison between an algorithm that utilizes the extended supernodes and one that utilizes standard supernodes are presented and discussed.

## 1 Introduction

In the last few years interior point methods (IPM) for linear programming (LP) have become increasingly popular. The growing experience of using these methods has shown that in general IPM algorithms complement and do not replace the established sparse simplex (SSX) algorithms.

One of the main differences between the IPM and the SSX is the average convergence rate. While the SSX average convergence rate is proportional to the number of constraints, the IPM convergence rate is almost independent of the growth in the problem size. As a consequence, IPM is increasingly seen as especially well suited for solving very large scale sparse LP problems.

*Department of Maths and Stats, Brunei, the University of West London, Uxbridge, Middlesex UBS 3PH, U.K. (Ron.Levkovitz@brunel.ac.uk)

The concentration of numerical work in relatively few steps and the need to solve very large LP problems makes the PM a good candidate for exploiting the power of parallel hardware [15]. The efforts of parallelizing IPM have so far concentrated on shared MIMD, distributed MIMD and vector computers. Karmarkar et al. [4] implemented the several IPMs on an Alliant MIMD computer. Bisseling et al. [3] adopted the dual affine IPM for a 20 X 20 transputer rack and achieved impressive speedup for a series of computationally difficult problems. Other implementations using superscaler, register and vector technology have also achieved a consistent speedup set against comparable serial implementations [13, 14, 8, 17].

In this paper we adopt the computationally intensive parts of IPM to a vector computer. We have implemented a specially adapted matrix multiplication algorithm and extend the Cholesky factorization algorithm to take advantage of dense processing within the sparse matrix.

The rest of the paper is organized as follows. In section 2 we present the primal dual predictor corrector IPM. In section 3 we analyze the computational structures of this IPM and provide summary profiling information. We use this information to illustrate why the IPM is well suited for parallelization. In section 4 we introduce our target hardware, the vector computer, and give some performance information. In section 5 we discuss alternative methods for the implementation of the symmetric matrix multiplication algorithm. The definition of the extended supernodes together with the Cholesky factorization that is based on it are presented in section 6. Finally, in section 7, we test and discuss our vector and sequential implementations on a range of industrial test problems.

## 2 The Predictor Corrector Interior Point Method

Of the many variants of the IPM that have been implemented, the primal dual algorithms in general, and the primal dual - predictor corrector algorithm in particular, are considered to be the most computationally attractive [16, 19]. Our implementation uses this predictor corrector variant and we describe below the algorithm.

### 2.1 The Predictor Corrector IPM

Consider the primal and dual LP problems in the standard form:

(primal) min $c^T x$ (1)

subject to $=$ $\geq$

(Dual) max $b^T y$ (2)

subject to $+$ $=$ $\geq$

A$\in \Re$ $\times$ $\in \Re$ $\in \Re$

Our aim is to calculate an optimal point ($x^*$, $y^*$, $z^*$) for this pair of non empty polyhedrons defined by their respective constraints (1) and (2).

Such a point satisfies the primal and dual constraints and the optimality criterion

$$- \quad = \quad = \qquad (3)$$

To solve the linear equation systems (l)-(2) and equation (3), we convert the constrained optimization problem to that of an unconstrained optimization. We first incorporate the non-negativity constraints in the objective function by introducing a logarithmic barrier function.

The new problems can be stated as

$$\min C^T x - \mu \sum^{n} \ln x_i \qquad (4)$$
$$\text{subject to } Ax = b$$

$$\max \quad + \quad \sum \qquad (5)$$

$$\text{subject to } A^T y + z = c$$

$$A \in \Re^{m \times n} \quad x,z,c \in \Re^n \quad b,y \in \Re^m$$

We further transform the problem to an unconstrained optimization problem by introducing the Lagrangian functions.

$$L(\text{Primal}) = c^T x - \mu \sum^{n} \ln x_i - y^T (Ax - b)$$

$$(6)$$

$$L(\text{Dual}) = b^T y + \mu \sum_{j=1}^{n} \ln z_j - x^T (A^T y + z - c)$$

The first order optimality conditions for the problems set out in (6) for given values of $\mu$ are

$$Ax - b = 0$$
$$A^T y + z - c = 0 \qquad (7)$$
$$XZe - \mu e = 0$$
$$x,z > 0$$

where $X$, $Z$ are diagonal matrices whose diagonals are $x,z$ and $e$ is a vector of all 1.

The search directions for the new points are derived from the conditions in (7). This is done by following the Newton direction, the Taylor polynomial direction, or some other method. Since the new point must satisfy the equations in (7). A simple way to derive the predictor corrector direction is to introduce a new point $(x + \Delta x, y + \Delta y, z + \Delta z x, )$. The new point must satisfy the equations in (7).A proper reduction in the value of the barrier parameter then gives us the desired improvement.

Substituting the new point in system (7) leads to the following set of equations.

$$A\Delta x = b - Ax$$
$$A^T\Delta y + \Delta z = C - A^T y + z \qquad (8)$$
$$\mathrm{X}\Delta\mathrm{Ze} + \mathrm{Z}\Delta\mathrm{Xe} = \mu\mathrm{e} - \mathrm{XZe} - \Delta\mathrm{X}\Delta\mathrm{Ze}$$

The system of equations in (8) contains nonlinear term, namely $\Delta\mathrm{X}\Delta\mathrm{Ze}$, hence we use a predictor corrector approach to solve it. We first calculate the predicting direction by ignoring the nonlinear terms and the $\mu$ term. Then we calculate $\mu$, according to the predicting direction and use it to calculate the correcting direction. The predicting direction obtained from (8) is

$$\Delta_p y = (ADA^T)^{-1} \ [dp - AD(z + d_D)]$$
$$\Delta p^X = D(A^T \Delta_p y - z + d_d) \qquad (9)$$
$$\Delta p^z = -(z + Zx^{-1}\Delta_p x)$$

Where D=$XZ^{-1}, dp = Ax - b,$ and $d_p = A^T y + z - c$.

The barrier parameter $p$. is calculated as a function of the duality gap after the maximum step is taken in the predicting direction.

$$\mu = f(C^T(x + \Delta_{\mathrm{p}} x) - b^T(y + \Delta_p y))$$

The correcting direction can then be calculated as.

$$= -$$

$$\Delta_{\mathrm{c}}\mathrm{x} = \mathrm{D}(\mathrm{A}^{\mathrm{T}}\Delta_{\mathrm{c}}\mathrm{y} - \mathrm{X}^{-1}(\mu\mathrm{e} - \Delta_{\mathrm{p}}\mathrm{X}\Delta_{\mathrm{p}}\mathrm{Ze})] \qquad (10)$$
$$\Delta_{\mathrm{c}} z = x^{-1}(\mu e - \Delta_p X\Delta_p Ze) - ZX^{-1}\Delta c^x)$$

In the k'th iteration:

(a)Create $H^k = AD^kA^T$

(b)Solve for $t_1^k : H^k t_1^k = r_1^k$

(c)Solve for $t_2^k : H^k t_2^k = r_2^k$

Figure 1: The main computational step of the IPM

The correcting and predicting directions are added to the current point to advance to the next point.

$$\begin{aligned} x &= x + \alpha p\ (\Delta_p x + \Delta_c x) \\ y &= y + \alpha D(\Delta_p y + \Delta_c Y) \\ z &= z + \alpha D\ (\Delta_p z + \Delta_c z) \end{aligned} \tag{11}$$

where $\alpha_{\mathrm{P}}$ and $\alpha_{\mathrm{D}}$ are primal and dual attenuation parameters $(0 < \alpha_P, \alpha_D < 1)$ that ensure that the new point is represented by a vector of strictly positive components.

## 2.2 Computational structure of the IPM algorithm

One of the fundamental reasons for the acceptance of the primal dual IPM is the polynomial worst case bound on the number of iterations. Indeed, if $L$ denotes the input size and $n$ the largest dimension of the constraint matrix $A$ then the algorithm converges in no more then $O(\sqrt{n}L)$ iterations [13. 1]. Practical implementations, however, show that the average case convergence rate is closer to $O(\log n)$. This means that only rarely will the number of iterations grow above 50-60 and most LP problems can be solved in 20-40 IPM iterations.

The computational work at each iteration of the predictor corrector IPM algorithm is concentrated in the solution of equations (9) and (10). It is easy to see that if $m$ and $n$ are of the same order then the repeated calculation of the matrix $ADA^T$ and its subsequent inversion dominate the computational process. The operations involved in these steps are illustrated in Figure 1.

The iterative process described above has the following properties:

1. If $A,D$ are of full rank then the matrix created in (a) is symmetric positive definite (SPD).

   Note that the matrices $H^k$ are only used to generate a search direction and thus, if any of them becomes numerically unstable we can fix it by replacing the appropriate diagonal elements. Therefore, we can safely assume that $\forall k, H^k$ is a positive definite matrix.

2. All the matrices $H^k$ have the same nonzero structure (in the following sections we refer

1. Cholesky factorization: Factorize $H — LL^T$
2. Set $L^T t = q$
3. Forward solve: Solve for $q$: $Lq = r$
4. Back solve : Solve for $t$: $L^T t = q$

Figure 2: The Cholesky algorithm

to these matrices simply as $H$).

3. The same matrix $H^k$ is used in the solution of steps (b) and (c)

There is a wide range of algorithms to solve systems of equations that are represented by a sparse SPD matrix (SSPD). In general, these algorithms can be divided into direct and iterative methods [10]. The above listed properties, especially the fixed structure of $H$ and the need to solve the system twice within a single iteration, strongly suggest the use of a direct method. In this paper we discuss only the direct solution method known as the Cholesky algorithm. The reader is referred to [18] and [22] for the application of iterative methods for the IPM. The steps of the Cholesky algorithm are summarized in Figure 2 and the main computational steps of the predictor corrector IPM algorithm using the Cholesky algorithm are set out in Figure 3.

## 3 The computationally intensive parts of IPM

Real life linear programming problems are usually very sparse. The symmetric matrix $ADA^T$, however, normally suffers some nonzero growth and can become much denser than the $A$ matrix. The Cholesky factor $L$, in turn, usually becomes even denser. For example, a single dense column in $A$ results in a fully dense $ADA^T$ and $L$. In practical implementations, substantial amount of work is spent on reducing the fill-in [7, 9, 13]. To begin with, dense columns are either split or calculated separately [25]. The symmetric matrix is then reordered to reduce the nonzero fill-in in the factorization. A common reordering strategy is the minimum degree algorithm. This method is used in our implementation of the predictor corrector IPM. However, depending on the nonzero structure of the original $A$ matrix, the amount of fill-in and the structure of the Cholesky factor can vary considerably. This affects the distribution of computational effort between the different parts of the algorithm as demonstrated in Tables1 and 2.

The problems we use come from two sources. The first source is the set of NETLIB models, the second is a set of industry generated LP problems. The models of the second set, named CAR*xx* and RAT1, originate from medical resolution enhancement of PET (positron emission

```
1. Initialize set $k = 0$, calculate $(x^k, y^k, z^k)$
2. Check for termination criteria
   if $(d_p^k < \in p)$ and $(d_D^k < \in D)$ and $((c^T x^k - b^T y^k)/(\|c^T x^k - b^T y^k\|) < \in G)$ then STOP
3. Factorize $L^k (L^k)^T = AD^k A^T$
4. Compute the predicting direction                    by using forward and back substitution
5. Compute the barrier parameter $\mu^K$
6. Compute the correcting direction $\Delta_c x^K, \Delta_c y^K, \Delta_C z^K$ by using forward and back substitution
7. Calculate $\alpha_p^k, \alpha_D^k$
8. Move to the new point
9. Set $k=k+l$, goto 2.
```

Figure 3: The IPM primal dual predictor corrector algorithm

| Model | Matrix | | | Matrix$AA^T$ | Cholesky | Iter. |
|---|---|---|---|---|---|---|
| | Rows | Cols | Nonz | Nonzeros | Nonzeros | |
| CAR2 | 400 | 1200 | 38890 | 58805 | 61411 | 15 |
| 25FV47 | 793 | 1849 | 10566 | 11715 | 32291 | 24 |
| PILOT | 1439 | 4655 | 42296 | 60977 | 205230 | 30 |
| CAR11 | 2025 | 6075 | 767804 | 1162527 | 1550510 | 24 |
| BNL2 | 2280 | 4442 | 14952 | 15688 | 89601 | 31 |
| RAT1 | 3136 | 9408 | 88267 | 219086 | 1251702 | 21 |
| CRE_A | 3422 | 7242 | 18142 | 24107 | 35924 | 29 |
| DFL001 | 6071 | 12230 | 35632 | 44169 | 1567825 | 50 |
| CAR4 | 16335 | 33652 | 63724 | 107696 | 169950 | 24 |
| CAR8 | 32768 | 67678 | 1183660 | 3276351 | 6280471 | 27 |

Table 1: Characteristics of the test problems

| Model | Build $ADA^T$ | Cholesky | Triag. solves | Other |
|---|---|---|---|---|
| CAR2 | 33.6% | 45.8% | 1.5% | 19.1% |
| 25FV47 | 13.0% | 51.4% | 5.3% | 30.3% |
| PILOT | 13.1% | 62.4% | 3.3% | 21.2% |
| CAR11 | 18.0% | 65.3% | 0.6% | 16.1% |
| BNL2 | 4.0% | 66.4% | 5.1% | 24.5% |
| RATI | 2.9% | 90.5% | 2.5% | 4.1% |
| CRE_A | 28.4% | 24.6% | 4.6% | 42.4% |
| DFL001 | 0.3% | 95.4% | 1.4% | 2.9% |
| CAR4 | 16.1% | 56.3% | 1.2% | 26.4% |
| CAR8 | 12.1% | 75.7% | 0.6% | 11.6% |

Table 2: Distribution of computational effort in a single sequential IPM iteration

tomography) images [24]. In choosing these models we have attempted to reflect the variety and size of real life LP models. The model CRE-A, an American Air Force airlift model [4], for instance, has relatively small nonzero fill-in during factorization while the model PILOT suffers a large amount of fill-in (see Figure 4).

The model PILOT represents a large class of problems whose Cholesky factor is fairly dense and requires a considerable amount of work. For these problems, large speed gains can be made by improving the efficiency of the matrix multiplication and factorization steps; either by improving the algorithms or by taking advantage of novel hardware features. In fact, some implementations of the IPM have been especially adapted to shared memory [23], distributed memory [3], and massively parallel [11] computers. All the above mentioned investigations are characterized by the close interaction between the implementation and the target architecture. An important aspect is that whichever method is used, some problems are less appropriate for the implementation than others. This is an inevitable consequence of the large variation in the models. In many cases, however, the matrices that suffer most from the advance methods are those that are easily solved in the naive sequential way. For a very sparse matrix with evenly distributed non zeros and limited fill-in (such as the problem CRE_A [4] ) the overhead in utilizing an advance architecture paradigm is almost always greater then the benefits. This type of problems can be recognized in advance and solved by using the serial implementation. Our main target is to speedup the solution process for those problems that are not solved in a reasonable time on standard sequential computers.

As the problem becomes more computationally intensive, the Cholesky factorization step becomes more dominant. There are several alternative schemes for the implementation of this step; the major ones being the column Cholesky, the row Cholesky, and the submatrix Cholesky [9]. Although the number of operations required by all these schemes is the same, they use different elimination sequences and therefore fit different computational approaches.

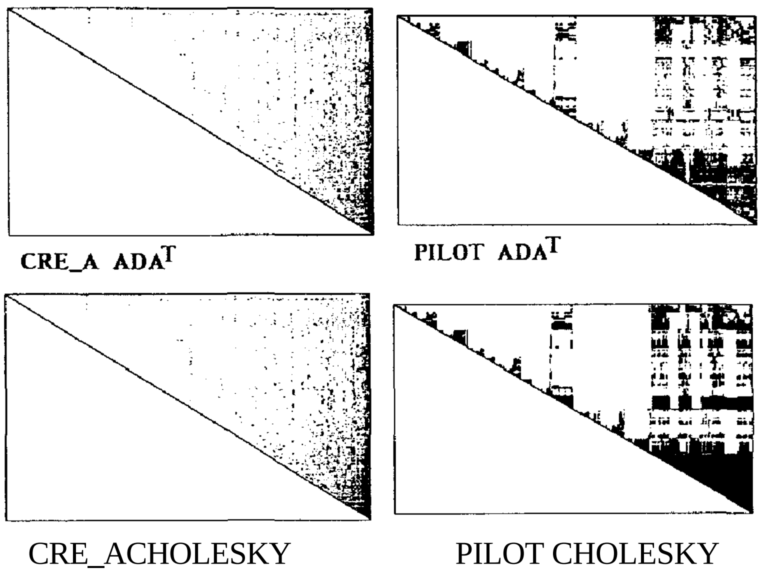

Figure 4: Symmetric matrix and Cholesky factor or the models CRE_A and PILOT

For our implementation, the column Cholesky algorithm proved to be the most suitable. It is convenient to represent the column Cholesky factorization as a sequence two procedures: *cmod(j, k)* (factorize column *j* using column *k*, equivalent to the BLAS routine *axpy* [5]) and *cdiv(j)* (scale column *j*, equivalent to the BLAS routine *scal* ).

$$cmod(j,k): \begin{bmatrix} h_{jj} \\ \cdot \\ \cdot \\ \cdot \\ h_{mj} \end{bmatrix} = \begin{bmatrix} h_{jj} \\ \cdot \\ \cdot \\ \cdot \\ h_{mj} \end{bmatrix} - l_{jk} \begin{bmatrix} l_{jk} \\ \cdot \\ \cdot \\ \cdot \\ l_{mk} \end{bmatrix} \qquad cdiv(j): \begin{bmatrix} l_{jj} \\ \cdot \\ \cdot \\ \cdot \\ l_{mj} \end{bmatrix} = \frac{1}{\sqrt{h_{jj}}} \begin{bmatrix} h_{jj} \\ \cdot \\ \cdot \\ \cdot \\ h_{mj} \end{bmatrix}$$

In Figure 5 we use the *cmod* and *cdiv* definitions to describe the column Cholesky algorithm.

As mentioned before, the elimination process starts with the restructuring and reordering of the symmetric matrix to reduce fill-in. The new elimination order determines the structure of the Cholesky factor. Thereafter, only the numerical values for the symmetric matrix and the Cholesky factor need to be calculated. These last calculations have to be carried out in very iteration.

In Figure 6 we list the actual operations involved in executing the computationally intensive parts of the IPM. These are grouped in two phases. The preprocessing phase contains all the operations that are done only once, that is, all operations that are concerned with structure only. The iterative phase contains all the operations that need to be done at every iteration, that is, the numerical operations.

```
do j — 1, m
  do k = 1 to j – 1
    if $l_{jk} \neq 0$ then
      cmod (j, k)
    endif
  enddo
  cdiv(j)
enddo
```

Figure 5: The column Cholesky factorization scheme

- Preprocessor Phase
  1. Create the symbolic structure of the symmetric matrix (Lower triangular only)
  2. Find a reordering of $H$ using the minimum degree algorithm
  3. Compute the symbolic factorization of the Cholesky matrix
  4. Analyze the symbolic Cholesky factor for special structures
- Iterative Phase
  1. Use the solution vectors of the previous iteration to compute the new diagonal matrix $D$
  2. Compute the numerical values of the symmetric matrix $H — ADA^T$
  3. Compute the numerical values of the Cholesky factor L using the column Cholesky scheme
  4. Compute the first right hand side vector
  5. Perform triangular forward and back solves to retrieve the predicting direction
  6. Compute the second right hand side vector
  7. Perform triangular forward and back solves to retrieve the correcting direction

Figure 6: Phases of computation

## 4 Vector processors and their properties

Within the various parallel architectures that emerged in the last few decades, the vector architecture is distinguished by its wide acceptance [12]. Nowadays, vector processors are used in the majority of supercomputers (e.g. CRAY-XMP), mainframes (e.g. IBM3090), and even in some workstations (e.g. DEC6000). Recently, single chip vector processors have become increasingly popular. These processors are designed for general use and are therefore much cheaper then their larger counterparts. Although these processors can execute any sequential code, a substantial speedup of an application is achieved only if the cache memory and vector capabilities are properly utilized.

### 4.1 utilizing the vector unit

In utilizing the vector unit, we have to consider the following points:

- The computation has to be rearranged such that it mostly consists of simple loops of elementary operations (e.g. multiplication loop).
- Indirect memory access should be minimized
- The use of conditionals within loops should be avoided
- Loops should be rearranged to remove data dependencies
- The vectorized loops should be as long as possible to reduce the effects of the startup overheads.

For example, the dense *axpy(x,y,α,)* operation [5] ($x = x + \alpha y$, $\alpha \in \Re, x, y \in \Re^n$) is ideal for vectorization. Sparse implementation of the *axpy is* less suitable due to the indirect memory access involved in the computation. To speed up dense computation and reduce indirect memory access, techniques such as loop unrolling are useful.

**Loop unrolling in dense computation:** consider for instance three separate *axpy* operation $axpy(x, y^1, \alpha_1)$, $axpy(x, y^2, \alpha_2)$, $axpy(x, y^3, \alpha_3)$, For every *axpy* operation, the elements of $x$ and $y^i$ have to be fetched from the memory, the elements of $y^i$ have to be scaled and added to $x$. All in all, the three *axpy* operations require $3(2n)$ fetch, $3n$ multiplication, $3n$ addition and $3n$ store operations. If, on the other hand, we where to construct a multiple *axpy* operation, that is $(x = x + \alpha_1 y^1 + \alpha_2 y^2 + \alpha_3 y^3)$ then, instead of fetching and storing $x$ for every *axpy* separately we fetch and store it only once thus saving $2n$ fetch and $2n$ store operations.

**Loop unrolling in sparse computation:** the advantage of loop unrolling become more evident when sparse operation is involved. Consider the sparse *axpy* operation where The sparse vectors are all packed. Due to the inability to vectorize operations where indirect memory addressing is involved, it is common to use *gather* and *scatter* operations. The *scatter* operation translates the packed vector to a temporary dense work area (a dense vector with explicit zero components). The *gather* operation re-packs and stores the work area. For every sparse *axpyS* operation, in addition to the dense operations, we also need to *scatter* the $y^i$ and $x$ vectors and to *gather* the $x$ vector. In the above case, loop unrolling will save two *gather* and two *scatter* operations. Further, if the vectors $y^1$, $y^2$, $y^3$ all have the same nonzero structure then it is possible to combine them by using dense *axpy.* Only the combined vector is scattered thus saving more *scatter* operations. It is obvious that the more vectors we can combine in such a way, the higher our savings are likely to be.

Operations such as *gather* and *scatter* are in most cases a part of standard primitives library provided by the hardware manufacturer. Other operations like *axpy* and *seal* are based on the basic linear algebra subprograms (BLAS) definitions [5] and provide optimized building blocks for the programmer. Arranging the application around these primitives and definitions usually results in a very efficient implementation.

## 4.2 Utilizing the cache unit

The cache is a small unit of high speed memory which maps the main memory. Every element that is fetched from the main memory to the processor is stored in the cache unit until a new element replaces it. If an element is required for computation while it resides in the cache then we have a cache hit. The cache is 2-5 times faster than the main memory and thus a large speedup can be achieved if elements are mostly fetched from the cache instead of the main memory. One of the main considerations of the algorithm designer is to maximize cache hits. This is done by blocking the data in such a way that enables the reuse of certain elements. Consider for example several sequential *axpy* operations where all the vectors are added to the same summation vector *x:*

1. $axpy(x,y^1,\alpha_1)$
2. $axpy(x,y^2,a_2)$
3. $axpy(x,y^3,a_3)$

If the vectors are short enough then the summation vector elements remain in the cache memory and can be reused. If, on the other hand, the vectors are too long, the elements are replaced and all the operations are done by using the main memory. In such a case, the vectors can be divided to subvectors of the appropriate length. For example, let the cache size be *n* then if we divide the vectors to 4 parts: $=[\quad]$ $=[\quad]$ the *axpy* operations can be scheduled as follows:

1. $axpy(x_1, y_1^i, \alpha_i)$, $i = 1,..,3,$
2. $axpy(x_2, y_2^i, \alpha_i)$, $i = 1,..,3$

3. $\text{axpy}(x_3, y_3^{\text{i}}, \alpha_{\text{i}})$ , i= 1,..,3,

*4.* $axpy(x_4, y_4^{\text{i}}, \alpha_{\text{i}})$ , $i$ = 1,..,3.

In this way, the elements of $x_1, x_2, x_3, x_4$ remain in the cache while needed and main memory access is minimized. Dividing vectors into subvectors that fit into the cache memory is always a good strategy for a serial processor. The vector processor, however, is more efficiently utilized when the vectors are the longest. Therefore, an extra care has to be taken not to divide the vectors to too small units that can increase the vectorization overhead.

In the next sections we show how the matrix multiplication and Cholesky factorization are scheduled to enable vectorized computation and to maximize cache hits.

## 5 Vector implementation of the symmetric matrix creation

The matrix $H = ADA^T$ is created in two stages, namely, symbolic and numeric multiplication. The symbolic multiplication is a simple algorithm that finds the off-diagonal nonzero in the symmetric matrix (lower triangular only) by using the following criterion:

$$h_{ij,i \neq j} \neq 0 \;\; iff \;\; \exists l : a_{il} \neq 0 \quad and \quad a_{jl} \neq 0 \tag{12}$$

After the symbolic symmetric matrix has been constructed the location of all the nonzeros is known. Thereafter, to calculate the actual value of a nonzero $h_{ij}$, the only operation that is needed is the multiplication of $(A_{i*}D)A_{j*}$ where $A_{i*}, A_{j*}$ denote the $i$'th and $j$'th rows of the constraint matrix $A$.

In most implementation reports, surprisingly little is said about this operation which typically takes around 15% of the iteration time. Bisseling et al. [3] and Marsten et al. [17] use a pre-calculated multiplication list for each nonzero in the symmetric matrix. As can be seen in equation (13), the couples $a_{il}a_{jl}$ (where both are nonzeros) are multiplied in advance and stored. Then, at every iteration, they are scaled by $d_{ll}^k$ and summed to create the nonzero

$h_{ij}^k$

$$h_{ij}^k = \sum_{l=1,..,n,\, a_{il}, a_{jl} \neq 0} [a_{il} a_{jl}] d_{ll}^k \tag{13}$$

The advantage of this method is the ability to perform the multiplication in a single dense loop with minimal indirect memory access. Although in this scheme a vector processor can be optimally utilized, it is rarely used due to the high memory requirements. These are especially high for LP problems whose $H$ matrix is fairly dense.

In our implementation we use two methods which attempt to utilize dense computation without the memory overhead of keeping the multiplication lists.

Consider a packed sparse row of the constraint matrix $A$. The packed row can be viewed as a collection of two dense vectors: a vector of real numbers which holds the numerical data and a vector of integers which holds the positions of the nonzero elements within the sparse row. The two most obvious ways to multiply these rows are listed below:

1. Fully dense computation: the row is scattered and multiplied with the matrix $D$. The result is multiplied using dense multiplication with the scattered rows that create a nonzero element $h_{ij}$ with

2. Fully sparse computation: a loop traces the collision points between and any row that need to be multiplied with it. When such a collision point is found, the elements are multiplied.

Both methods have serious drawbacks. Typically, the nonzeros account for only a fraction of the row size and thus, the first method can have large overheads in scattering the vectors and in redundant multiplications. The second method, on the other hand, results in many conditional operations. If the vectors have many nonzeros few collision points, the overheads will be very large. In addition, due to the conditional computation, this method is particularly bad for vector implementation.

Instead, we implement the following two alternative schemes:

1. For the nonzeros of $H_{i*}$ the row $A_{i*}$ is scattered and multiplied with $D$. The rows $A_{i*}$ that are multiplied with it are not scattered; instead, the loop passes on the location vector of the packed rows and multiply every element in the packed row with its counterpart in the scattered row regardless if it was originally a nonzero.

   Although this method results in redundant multiplications, they are limited to the number of the nonzeros in the packed row. For the whole multiplication, only $n$ scatter operations are needed. The method completely avoids using conditional computation but still uses indirect memory access.

2. For the nonzero $h_{ij}$ the row $A_{i*}$ is scattered and multiplied with $D$. It is then gathered again but this time according to the index vector of the row This allows dense multiplication between the vectors to be performed. The scheme utilizes dense computation and completely removes the need for indirect memory access. The price for this is the additional gather operations. The number of multiplication, however, it the same as in the first method.

In Table 7, section 7, we compare the performance of these two schemes on a serial and vector processors for several NETLIB and industrial problems.

æ

# 6Vectorized Cholesky factorization

Our implementation of the column Cholesky factorization algorithm is targeted towards utilizing dense computation and minimizing main memory usage. We attempt to reorganize the elimination such that most operations take place within large and dense blocks. If large enough dense blocks are identified, a multiple dense *cmod* (*axpy*) operation can be carried out instead of using several sparse operations. In addition, the dense vectors can be partitioned to blocks that fit the cache size (see section 3).

An important feature in increasing the scope of dense computation is the identification of structurally indistinguishable columns (supernodes).

**Definition 1: Supernodes** *For* ***a*** *given SSPD matrix ordering, a supernode is made of a group of columns that in the Cholesky factor L create a dense triangular block just below the diagonal and have the same nonzero structure elsewhere.*

Here, we extend the above definition in to further increase the scope of dense computation.

**Definition 2: Extended Supernodes** *For a given SSPD matrix ordering the extended supernode associated with the column $L_{*j}$ is made of all columns $L_{*j}$, $i < j$ whose nonzero structures in the Cholesky factor in positions j,. .., m is the same as that of the column $L_{*j}$. In other words, the vectors $[l_{ji},\ldots,l_{mi}]$ and $[l_{jj},\ldots l_{mj}]$ have the same nonzero structure.*

It is clear that if the Cholesky factor $L$ is fully dense then there is only a single supernode which includes all the columns. In such case, the extended supernode of any column $L_{*j}$ is simply the group of all preceding columns. In sparse matrices, however, the extended supernodes are larger than the supernodes. On the negative side, the extended supernodes are defined per column and therefore the effort of finding and storing them is higher.

The Cholesky factor in Figure 7 demonstrates the difference between supernodes and extended supernodes. Columns 1,2 and 3 can be grouped into a supernode as well as 5,6 and 7. For column 6, however, all preceding columns are members of the extended supernode,

Supernodes and extended supernodes are used in a similar fashion during the Cholesky factorization.

1. Assume that the column $L*j$ is currently being factorized. If $L*j$ is factorized by a column $L_{*j}$ which is a member of the same (extended) supernode then the *cmod(j, i)* operation can be done using dense mode.

2. If the column $L_{*k}$ factorizes $L_{*j}$ then all the columns $L_{*t}$, $t < j$ in $L_{*k}$'s (extended) supernode must also factorize it. The contributions of all these columns can be collected together and added to $L_{*j}$ in one *cmod* step. These columns all have the same nonzero structure from the j'th place and onwards. Thus, the number of indirect memory operations is reduced and loop unrolling and cache hit techniques can be used (see section 3).

$$\begin{array}{ccccccc} \times &&&&&& \\ \times & \times &&&&& \\ \times & \times & \times &&&& \\ &&& \times &&& \\ \times & \times & \times && \times && \\ \times & \times & \times & \times & \times & \times & \\ \times & \times & \times & \times & \times & \times & \times \\ \hline 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \tag{14}$$

Figure 7: The Cholesky Factor

**Dense Window:** a commonly used approach for utilizing dense computation is the creation of a dense window. Here, we take advantage on the fact that most fill-in usually takes place in the lower part of the Cholesky factor creating an almost dense matrix (see for example Figure 1 in section 2). In such a case, when the percentage of nonzeros per column grows above a certain value (50-70%, say) then the rest of the matrix is treated as fully dense.

Consider, for example, the following H matrix whose lower part F is dense:

$$\begin{bmatrix} H_{11} & H_{21}^{T} \\ H_{21} & F_{22} \end{bmatrix} \tag{15}$$

The Cholesky factorization of *H* can be scheduled as follows:

- Sparse
  (i) Factorize $H_{11} = L_{11} \; L_{11}^{T}$
  (ii)Update $L_{21} = H_{2l}(L_{11}^{-1})^{T}$

- **Sparse-Dense**

  (iii) Update $\hat{F}_{22} = F_{22} - L_{21}L_{21}^{T}$

- **Dense**

  (iv) Factorize $\hat{F}_{22} = L_{22}L_{22}^{T}$

The factorization of the dense part (step (iv)) can be carried out either by using techniques similar to those used for the supernodes or by using a hardware optimized dense BLAS Cholesky factorization routine [6].

We note, however, that for the creation of a dense window we sometimes need to add explicit zeros to the lower part of the Cholesky factor $\hat{L}_{22}$. These operations alters the structure of the Cholesky factor and can reduce the scope of utilizing the (extended) supernodes.

There are many different methods for speeding up the Cholesky factorization algorithm, see for example [9, 7] and [2]. Most methods offer speedup of the actual factorization while causing a slowdown or overheads in other parts. For this reason, they are not beneficial for all the problems. The two methods we present below fall into this category. The first method is an implementation of the Cholesky algorithm using supernodes. This implementation requires some extra storage space and perform more operations hence it is clearly unfavourable when the number and size of supernodes is small. The second implemenation that utilizes the extended supernodes requires larger storage but does not perform more operations. Both implementations, however, are beneficial for a large class of problems and computers (see section 7).

## 6.1 Cholesky factorization using Supernodes

The identification of supernodes and extended supernodes is done after the order of elimination (and therefore the structure of the Cholesky factor) has been determined.

For the identification of supernodes we use the following criterion:
*if* $l_{j+1,j} \neq 0$ *and* $nz\,([l_{j+2,j}, \ldots l_{m,j}]) = nz([l_{j+2,j+1}, \ldots, l_{m,j+1}])$ *then* $L_{*j}, L_{*,j+1}$ *are members of the same supernode*
where *nz(vec]* denotes the number of nonzeros in the vector.

The supernode information is kept in an array of integers *super_n* of the size 4*m* bytes. If the columns whose numbers are $i_1 < i_2 < i_3,.., < i_k$ are members of a supernode then *super_n*($i_i$) = *super_n* ($i_2$ ) =,…,= *super-n*($i_k$) = $i_1$ (note that a single column is treated as a supernode of length one).

It is easy to see that a single pass on the data structure is sufficient for identifying all the supernode. This takes at most $O(m)$ operations.

During the Cholesky algorithm, when computing the column $L_{*j}$, we build an elimination stack that contains the columns needed for the elimination. The columns in the stack are stored in a descending order (the highest on top). In such a way, when a column is factorized, it is enough to find the beginning of its supernode in the *super_ n* array. The columns in a supernode are always consecutive and therefore the preceding columns in the supernode can be removed from the stack without further checks. The actual elimination can therefore take place by using a dense, multiple *cmod* operation as described in section 4.

## 6.2 Cholesky factorization using extended supernodes

The identification of an extended supernode is done in a similar way to the identification of the supernodes.

Let $L_{*j}$ be the current column then $L_{\neq i}$ is a member of the extended supernode if
$((i < j$ and $L_{j,i} \neq 0$ and $nz\,([l_{j,i}, \ldots, l_{m,i}]) = nz\,([l_{j,j}, \ldots, l_{m,j}]))$

The extended supernode information is held in an array of integers *super_ x* of the size *4lnz* bytes where *Inz* is the number of nonzeros in the Cholesky factor. In addition, two pointer arrays *x_group* and *s_group* of the size *4(m+l)* bytes each are required. The array *x_group* stores the pointers to the beginning of the extended supernode section and the array *s_group* to the beginning of the single columns section. Thus, for the column $L_{*j}$, the extended supernode members are stored in the positions *x_group*($j$),.,*x_group*($j$ + 1) - 1 and the columns that factorize $L_{*j}$ but are not members of the extended supernodes are stored in the positions *s_group(j)*,.,*S_group*($j$ + 1) - 1 of the same array. The complexity of constructing the extended supernode data structure is at most $O(lnz + 2m)$.

During the elimination process, the members of the extended supernode of factorize the current column using multiple dense *cmod*. If a column is not a member of the extended supernode then sparse elimination must take place. However, all the columns that are members of the extended supernode of also need to factorize . The contributions of these columns are combined and only a single sparse *cmod* is performed. Note that the method of keeping explicit elimination list removes the need for construction the elimination stack and therefore saves operations.

## 7 Experimental results

The ideas detailed in the previous section were added to our existing IPM code [13]. The test are performed on two hardware platforms. An i486/50MHz 16Mbyte PC and an i860/40MHz 32Mbyte vector computer. On both computers, the programs were compiled and run using the Microway NDP environment and Fortran 77 [21].

The i860 is representative of a new generation of single chip vector processors. It is a RISC processor with vector capabilities. In addition, it has an on-chip cache unit. The processor vector and cache capabilities are utilized via a vector primitive library [20]. This library is based on the BLAS definitions and includes hardware optimized gather and scatter.

In Table 4 we present the timings of the symmetric matrix multiplication schemes on the i860 vector computer. The first scheme uses indirect memory access and sparse/dense multiplication, the second scheme uses scatter-gather and dense multiplication. The second and third columns in the table give the number of rows (columns) in the symmetric matrix and the average number of nonzeros per row. The forth and fifth columns give the execution time in seconds of the two schemes. It is easy to see that the first scheme is better when the average number of nonzeros per row (column) falls below a certain number (around 8 in our case). This behaviour co-insides with the fact that utilizing the vector capabilities for less then 10 elements in a loop is likely to result in a slowdown instead of speedup [20]. It is, however, inefficient to check the length of any multiplication loop; the 'if question is as expensive as the initialization of the vector array. On the other hand, the average number of nonzeros per row can be calculated in the initialization phase and the algorithm can be

| Model | Matrix $A$ | | | Matrix $AA^T$ | Cholesky factor |
|---|---|---|---|---|---|
| | Rows | Columns | Nonz | Nonz | Nonz |
| CAR2 | 400 | 1200 | 38890 | 58805 | 61411 |
| 25FV47 | 793 | 1849 | 10566 | 11715 | 32291 |
| PILOT | 1439 | 4655 | 42296 | 60977 | 205230 |
| BNL2 | 2280 | 4442 | 14952 | 15688 | 89601 |
| RAT1 | 3136 | 9408 | 88267 | 219086 | 1251702 |
| CRE_A | 3422 | 7242 | 18142 | 24107 | 35924 |
| DFL001 | 6071 | 12230 | 35632 | 44169 | 1567825 |
| CAR4 | 16335 | 33652 | 63724 | 107696 | 169950 |

Table 3: Characteristics of the test problems

| Model | $AD\,A^T$ | | Timings on i860 (sec) | |
|---|---|---|---|---|
| | Rows | Nz/row(Av.) | Scheme 1 | Scheme 2 |
| CAR2 | 400 | 147.01 | 4.15 | 1.65 |
| 25FV47 | 793 | 22.33 | 0.28 | 0.32 |
| PILOT | 1439 | 42.2 | 2.52 | 1.83 |
| BNL2 | 2280 | 6.8 | 0.26 | 1.01 |
| RAT1 | 3136 | 69.86 | 4.76 | 5.33 |
| CRE_A | 3422 | 7.04 | 0.89 | 2.55 |
| DFL001 | 6071 | 7.27 | 1.37 | 6.74 |
| CAR4 | 16335 | 6.58 | 4.28 | 43.90 |

Table 4: Symmetric matrix multiplication

chosen according to this number. Although this approach does not guarantee a speedup, it is likely to prevent a serious slowdown.

In Table 5 we present the breakdown of the elimination to dense and sparse elimination for supernodes and extended supernodes. The column named 'cmod' gives the total number of 'cmod' operations in the factorization. For both supernodes and extended supernodes, the column named 'single sparse' gives the number of sparse cmod and the column named 'total dense' gives the number of dense cmod operations. The column marked 'Multiple sparse' gives the number of sparse cmod operations that were done by a combined elimination vector. This vector which results from collecting the contributions of columns in the (extended) supernodes is an overhead created by using dense computation.

From Table 5 it is clear that the contribution of extended supernodes is not only in increasing the scope of dense computation but also in removing the overheads of using multiple sparse cmod operations. In addition, the increase in dense computation also contributes to an increased scope of loop unrolling.

In Tables 6-9 we present the timing results for the supernode and extended supernodes

| | | Supernodes | | | Extended Supernodes | | |
|---|---|---|---|---|---|---|---|
| Model | Total cmod | Single sparse | Mutiple sparse | Total dense | Single sparse | Mutiple sparse | Total dense |
| CAR2 | 61011 | 39690 | 0 | 21321 | 38680 | 0 | 22331 |
| 25FV47 | 33498 | 5442 | 3195 | 28056 | 5401 | 2913 | 28097 |
| PILOT | 203791 | 27229 | 8481 | 176562 | 27156 | 7000 | 176635 |
| BNL2 | 87336 | 14372 | 4719 | 72964 | 14049 | 4267 | 73287 |
| RAT1 | 1242566 | 61034 | 21009 | 1187532 | 61005 | 20307 | 1187561 |
| CRE_A | 32512 | 18811 | 3544 | 13701 | 18600 | 3258 | 13912 |
| DFL001 | 1561754 | 168032 | 46395 | 1393722 | 163149 | 33324 | 1398605 |
| CAR4 | 153566 | 30732 | 1992 | 122834 | 27684 | 1839 | 125882 |

Table 5: Dense and sparse cmod operations

implementation on i486 and i860 computers ( the models RATl and DFL001 could not be solved on the i486 due to lack of memory). From the tables we conclude that if the models involved are fairly dense, the differences between implementations utilizing supernodes and extended supernodes are not very significant. The extended supernodes implementation re-quires more memory and it is therefore obvious that standard supernodes are preferable in this case. For sparser matrices, however, the static elimination list and the increased dense computation account for large reductions in the execution time. This is even more so on the vector computer where dense computation is efficiently utilized. Hence, in problem like CAR4 we experience a 17 fold reduction in execution time. The sparsity of the Cholesky factor is calculated during the initialization phase and it is therfore possible to determine which elim- ination method to use beforehand. This decision is made on the basis of the density of the Cholesky factor and the amount of the available memory. If the matrix is large and sparse and the amount of available memory is sufficient then the extended supernode scheme is used. Otherwise, standard supernodes are utilized.

**Acknowledgment**

The author gratefully acknowledges the valuable help of Prof. G. Mitra and Dr. H. Hafsteins-son in writing this paper. We also like to thank Dr. T. Jones and Mr. C. Tong of the MRC Cyclotron Unit of Hammersmith Hospital for supplying the PET models and for working closely with us in the solution of those problems.

|  | General Timings (Sec) |  |  | Iteration Timings (Sec) |  |  |
|---|---|---|---|---|---|---|
| Model | time | Iter. | Init | Build $ADA^T$ | Chol | Other |
| CAR2 | 188.179 | 17 | 4.640 | 4.449 | 5.710 | 0.596 |
| 25FV47 | 55.749 | 25 | 0.822 | 0.281 | 1.480 | 0.436 |
| PILOT | 1006.292 | 34 | 8.332 | 2.363 | 25097 | 1.891 |
| BNL2 | 336.308 | 34 | 4.696 | 0.281 | 8.562 | 1.191 |
| RAT1 | - | - | - | - | - | - |
| CRE_A | 180.210 | 34 | 7.956 | 0.707 | 3.242 | 1.117 |
| DFL001 | - | - | - | - | - | - |
| CAR4 | 1715.761 | 23 | 121.095 | 4.449 | 59.820 | 5.064 |

Table 6: Timings breakdown using supernodes on the i486

|  | General Timings (Sec) |  |  | Iteration Timings (Sec) |  |  |
|---|---|---|---|---|---|---|
| Model | time | Iter. | Init | Build $ADA^T$ | Chol | Other |
| CAR2 | 237.339 | 17 | 5.839 | 4.226 | 8.679 | 0.72 |
| 25FV47 | 54.046 | 25 | 1.069 | 0.222 | 1.429 | 0.434 |
| PILOT | 1023.749 | 34 | 5.151 | 2.359 | 25.601 | 0.224 |
| BNL2 | 296.097 | 34 | 6.244 | 0.269 | 7.359 | 0.897 |
| RAT1 | - | - | - | - | - | - |
| CRE_A | 102.546 | 34 | 11.438 | 0.769 | 0.820 | 1.09 |
| DFL001 | - | - | - | - | - | - |
| CAR4 | 856.507 | 23 | 153.238 | 4.613 | 21.750 | 17.137 |

Table 7: Timings breakdown using extended supernodes on the i486

|  | General Timings (Sec) |  |  | Iteration Timings (Sec) |  |  |
|---|---|---|---|---|---|---|
| Model | time | Iter. | Init | Build $ADA^T$ | Chol | Other |
| CAR2 | 125.769 | 17 | 5.835 | 4.15 | 2.539 | 0.360 |
| 25FV47 | 43.77 | 30 | 1.390 | 0.28 | 0.799 | 0.320 |
| PILOT | 328.229 | 33 | 9.343 | 2.52 | 6.260 | 0.866 |
| BNL2 | 186.009 | 40 | 5.436 | 0.26 | 3.580 | 0.564 |
| RAT1 | 1425.278 | 25 | 64.998 | 4.76 | 46.489 | 3.162 |
| CRE_A | 256.549 | 37 | 9.092 | 0.89 | 5.139 | 0.659 |
| DFL001 | 6055.889 | 48 | 137.825 | 1.37 | 118.010 | 3.913 |
| CAR4 | 2334.960 | 26 | 110.411 | 4.28 | 78.589 | 2.69 |

Table 8: Timings breakdown using supernodes on the i860

|  | General Timings (Sec) |  |  | Iteration Timings (Sec) |  |  |
|---|---|---|---|---|---|---|
| Model | time | Iter. | Init | Build $ADA^T$ | Chol | Other |
| CAR2 | 124.159 | 17 | 6.275 | 4.15 | 2.410 | 0.374 |
| 25FV47 | 29.711 | 26 | 2.170 | 0.28 | 0.520 | 0.280 |
| PILOT | 298.229 | 34 | 10.578 | 2.52 | 5.070 | 0.870 |
| BNL2 | 90.510 | 34 | 7.253 | 0.26 | 1.620 | 0.567 |
| RAT1 | 1326.090 | 25 | 70.330 | 4.76 | 42.329 | 3.143 |
| CRE_A | 80.070 | 34 | 12.792 | 0.89 | 0.430 | 0.658 |
| DFL001 | 5003.736 | 48 | 151.512 | 1.37 | 95.811 | 3.907 |
| CAR4 | 439.959 | 25 | 148.856 | 4.28 | 4.660 | 2.704 |

Table 9: Timings breakdown using extended supernodes on the i860

## References

[1] I. ADLER, N. K. KARMARKAR, M. G. C. RESENDE, AND G. VEIGA, *An implementation of Karmarkar's algorithm for linear programming,* Mathematical Programming, 44 (1989), pp. 297-335. Errata in *Mathematical Programming*, 50:415, 1991.

[2] C. ASHCRAFT, R. GRIMES, J. LEWIS, B. PEYTON, AND H. SIMON, *Progress in spars matrix methods for large linear systems on vector supercomputers,* Internal. J. Supercomp. Appl., 1 (1987), pp. 10-30.

[3] R. H. BISSELING, T. M. DOUP, AND L. D. J. C. LOYENS, *A parallel interior point algorithm for linear programming on a network of 400 transputers,* Annals of Operations Research, 43 (1993).

[4] W. CAROLAN, J. HILL, J. KENNINGTON, S. NIEMI, AND S. WICHMANN, *An empirical evaluation of the KORBX algorithms for military airlift applications,* Operations Research, 38 (1990), pp. 240-248.

[5] J. DONGARRA, J. DUCROZ, S. HAMMARLING. AND R. HANSON, *An extended set of fortran basic linear algebra subprograms,* ACM Trans. Math. Software, 14 (1988), pp. 1-17.

[6] J. J. DONGARRA, J. R. BUNCH, C. B. MOLER, AND G. W. STEWART, *LINPACK User's Guide*, SIAM, Philadelphia, 1979.

[7] I. DUFF, A. ERISMAN, AND J. REID, *Direct Methods for Sparse Matrices,* Oxford University Press, 1989.

[8] J. J. H. FORREST AND J. A. TOMLIN, *Implementing interior point linear programming methods in the Optimization Subroutine Library,* IBM Systems Journal, 31 (1992), pp. 26-38.

[9] A. GEORGE AND J. W. H. LIU, *Computer Solutions of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[10] G. H. GOLUB AND C. F. V. LOAN, *Matrix Computations,* North Oxford Academic, 1983.

[11] H. HAFSTEINSSON, R. LEVKOVITZ, AND G. MITRA, *Solving large scale linear programming problems using an interior point method on a massively parallel SIMD computer,* Technical Report TR/05/93, Department of Maths and Stats, Brunel University, Uxbridge, Middlesex UBS 3PH, 1993.

[12] R. W. HOCKNEY AND C. R. JESSHOPE, *Parallel Computers 2: Architecture, Programming and Algorithms,* IOP Publishing, 1988.

[13] R. LEVKOVITZ, *An Investigation of Interior Point Methods for Large Scale Linear Programs: Theory and Computational Algorithms,* PhD thesis, Brunel, The University of West London, 1992.

[14] R. LEVKOVITZ AND G. MITRA, *Solution of large sparse symmetric equations on a transputer network,* in Proceedings of the Third International Conference on Applications of Transputers, IOS Press, 1991, pp. 105-110.

[15] –,*Solution of large-scale linear programs: A review of hardware, software and algorithmic issues,* in Optimization in Industry, T. A. Ciriani and R. C. Leachman, eds., John Wiley & Sons, 1993, pp. 139-171.

[16] I. J. LUSTIG, R. E. MARSTEN, AND D. F. SHANNO, *Interior point methods: Computational state of the art,* Technical Report, School of Engineering and Applied Science, Dept. of Civil Engineering and Operations Research, Princeton University, Princeton, NJ 08544, USA, December 1992. Also available as RUTCOR Research Report RRR 41-92, RUTCOR, Rutgers University, New Brunswick, NJ, USA. To appear in *ORSA Journal on Computing.*

[17] R. E. MARSTEN AND D. F. SHANNO, *Interior point methods for linear programming: Ready for production use,* Workshop at the ORSA/TIMS Joint National Meeting in Philadelphia, PA, USA, School of Industrial and System Engineering, Georgia Institute of Technology, Atlanta, GA 30322, USA, October 1990.

[18] S. MEHROTRA, *Implementation of affine scaling methods: Approximate solutions of systems of linear equations using preconditioned conjugate gradient methods,* ORSA Journal on Computing, 4 (1992), pp. 103-118.

[19] —— ,*On the implementation of a primal-dual interior point method,* SIAM Journal on Optimization, 2 (1992), pp. 575-601.

[20] MICRO WAY CORP., *i860 Microprocessor Vector Primitive Library Reference Manual,* 1990.

[21] ——, *NDP 860 Tool Chain Reference Manual,* 1991.

[22] M. G. C. RESENDE AND M. H. WRIGHT, *Sixty-six attend interior methods workshop at Asilomar,* SIAM News, 23 (1990), p. 9.

[23] M. J. SALTZMAN, *Implementation of an interior point LP algorithm on a shared-memory vector multiprocessor*, in Operations Research and Computer Science: New Developments in Their Interfaces, O. Balci, R. Sharda, and S. A. Zenios, eds., Pergamon Press, Oxford, UK, 1992.

[24] C. TONG, S. GROOTOONK, H. BYRNE, T. SPINKS, A. LAMMERTSMA, AND T. JONES, *Positron emission tomography: Recovery of resolution by Finite Elements method,* The Journal of Nuclear Medicine, 34 (1993), pp. 26P-27P.

[25] R. J. VANDERBEI, *Splitting dense columns in sparse linear systems,* Linear Algebra and Its Applications, 152 (1991), pp. 107-117.