TR/05/93 **August 1993**

SOLVING LARGE SCALE LINEAR PROGRAMMING PROBLEMS USING AN INTERIOR POINT METHOD ON A MASSIVELY PARALLEL SIMD COMPUTER

Hjálmtýr Hafsteinsson, Roni Levkovitz
Gautam Mitra

Solving Large Scale Linear Programming Problems Using an Interior Point Method on a Massively Parallel SIMD Computer

Hjálmtỳr Hafsteinsson* Roni Levkovitz† Gautam Mitra‡

Abstract

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

We describe an implementation of the predictor corrector IPM algorithm on MasPar, a massively parallel SIMD computer. At the heart of the implementation is a parallel Cholesky factorization algorithm for sparse matrices. Our implementation uses a new scheme of mapping the matrix onto the processor grid of the MasPar, that results in a more efficient Cholesky factorization than previously suggested schemes.

The IPM implementation uses the parallel unit of MasPar to speed up the factorization and other computationally intensive parts of the IPM. An important part of this implementation is the judicious division of data and computation between the front-end computer, that runs the main IPM algorithm, and the parallel unit. Performance results on standard industrial test problems are presented and discussed.

---

*Department of Computer Science, University of Iceland, Reykjavik, Iceland (hh@rhi.hi.is). This work was performed while the author was visiting the Department of Mathematics and Statistics, Brunei University

†Department of Mathematics and Statistics, Brunel University, Uxbridge, Middlesex UBS 3PH, U.K. (Ron.Levkovitz@brunel.ac.uk)

‡Department of Mathematics and Statistics, Brunel University, Uxbridge, Middlesex UBS 3PH, U.K. (Gautam.Mitra@brunel.ac.uk)

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 invariant to the growth in the problem size. As a consequence, the IPM is considered to be well suited for solving very large sparse LP problems. For a discussion of some of the research issues we refer the reader to [12].

The concentration of numerical work in relatively few steps and the need to solve very large LP problems makes the IPM a good candidate for exploiting the power of parallel hardware. The efforts of parallelizing IPM have so far concentrated on shared MIMD [17], distributed MIMD [11] and vector computers. Carolan et al. [3] implemented several IPMs on an Alliant MIMD computer. Bisseling et al. [1] adapted 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 superscalar, register and vector technology have also achieved consistent speedups compared to similar serial implementations [5, 10, 15].

In this paper we describe our approach of deploying the computationally intensive parts of IPM to a massively parallel SIMD computer. We have implemented a matrix multiplication algorithm and an algorithm for solving triangular systems on the SIMD computer. We also make use of the Cholesky factorization algorithm of Manne and Hafsteinsson [14]. We show that for a large number of LP problems the SIMD implementation is a viable alternative and that by a simple preanalysis of matrix structure and nonzero statistics, these problems, which are suitable for SIMD processing, can be identified in advance.

The rest of the paper is organized as follows. In section 2 we present the primal dual - predictor corrector IPM. We also analyze the computational structure of this IPM and provide summary profiling information. We use this information to illustrate why the IPM is well suited for parallelization. In section 3 we introduce our target hardware, the MasPar MP-2 SIMD computer and give some performance information. In section 4 we describe the MasPar implementation, and in particular present the symmetric matrix multiplication, the Cholesky factorization, and the triangular solution algorithms and

explain how data movement between the parallel unit of the MasPar and the front-end computer is handled. In section 5 we profile a set of difficult test models and compare the solution times of the MasPar to those of a top of the range Sparc 10 computer. We further use these results to develop a general criterion that enables us to decide which models should be solved on the MasPar. Finally, in section 6, we discuss our results and future extensions to the algorithm.

2 Interior Point Methods

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 [13, 16]. 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:

$$\begin{aligned} &\text{(Primal)} && \min\ c^T x && (1)\\ &&& \text{subject to } Ax = b, x \geq 0 \\ &\text{(Dual)} && \max\ b^T y && (2)\\ &&& \text{subject to } A^T y + z = c, z \geq 0 \\ &&& A \in \Re^{mxn}\ x,z,c \in \Re^n\ y,\ b\ \in \Re^m . \end{aligned}$$

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

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

$$c^T x^* - b^T y^* \ = \ (x^*)^T z^* = 0 \qquad (3)$$

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

$$\min \ c^T x - \mu \sum_{j=1}^{n} \ln x_j \tag{4}$$

subject to $Ax = b$

$$\max b^T y + \mu \sum_{j=1}^{n} \ln z_j \tag{5}$$

subject to $A^T y + z = c$

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

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

$$\begin{aligned} L(\text{Primal}) &= c^T x - \mu \sum_{j=1}^{n} \ln x_j - y^T (Ax - b) \\ L(\text{Dual}) &= b^T y + \mu \sum_{j=1}^{n} \ln z_j - x^T (A^T y + z - c) \end{aligned} \tag{6}$$

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

$$\begin{aligned} Ax - b &= 0 \\ A^T y + z - c &= 0 \\ XZe - \mu e &= 0 \\ x, z &> 0 \end{aligned} \tag{7}$$

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 $(x+\Delta x, y + \Delta y, z + \Delta z)$. 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.

$$\begin{aligned} A\Delta x &= b\text{-}Ax \\ A^T \Delta y + \Delta z &= c\text{-}A^T y + z \\ X \Delta Ze + Z \Delta Xe &= \mu e - XZe - \Delta X \Delta Ze \end{aligned} \tag{8}$$

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

$$\begin{aligned} \Delta_p y &= (ADA^T)^{-1}[d_P - AD(z + d_D)] \\ \Delta_p x &= D(A^T \Delta_p y - z + d_d) \\ \Delta_p z &= -(z + ZX^{-1} \Delta_{pX}) \end{aligned} \tag{9}$$

where $D = XZ^{-1}$, $dp = Ax - b$ and $d_D - A^T y + z\text{-}c$.

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

$$\mu = f(c^T(x + \Delta_{pX}) - b^T(y + \Delta_p y))$$

The correcting direction can then be calculated as

$$\begin{aligned} \Delta_C y &= -(ADA^T)^{-1} X^{-1}(\mu e - \Delta_p X \Delta_p Ze) \\ \Delta_c x &= D[A^T \Delta_c y - X^{-1}(\mu_e - \Delta_p x \Delta_p Ze)] \\ \Delta_C z &= X^{-1}(\mu e - \Delta_p X \Delta_p Ze) - ZX^{-1} \Delta_c x) \end{aligned} \tag{10}$$

The correcting and predicting directions are added to the current point to advance to advance 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 α_p and α_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 $O(\sqrt{nL})$ iterations [9]. 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. Here, it is significant to observe that if the matrix A is of full rank then the matrix ADA^T is symmetric positive definite. Further, the nonzero structure of the matrix remains invariant throughout the iterative process. Since the same matrix is required in both (9) and (10), it is also natural to use a solution method that allows us to solve the second system of equations without repeating the calculation.

There are many direct and iterative methods for solving a symmetric positive definite system of equations [7]. We have adopted one of the more popular approaches, namely, the Cholesky factorization. The main computational steps of the predictor corrector IPM algorithm with the Cholesky factorization are set out in Figure 1.

2.3 Practical Considerations

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

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* $((c^T x^k - b^T y^k)/(||b^T y^k||) < \in_{gap})$ *then*

   STOP
3. *Factorize* $L^k (L^k)^T = AD^k A^T$
4. *Compute the predicting direction* $(\Delta_p x^k, \Delta_p y^k, \Delta_p z^k)$
5. *Compute the barrier parameter* μ^k
6. *Compute the correcting direction* $(\Delta_c x^k, \Delta_c y^k, \Delta_c z^k)$
7. *Calculate* α_P, α_D
8. *Move to the new point*
9. *Set k = k+1, goto 2.*

Figure 1: The IPM primal dual - predictor corrector algorithm

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 [4, 6, 9]. To begin with, dense columns are either split or calculated separately [19]. 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 Tables 1 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 tomography) images [18]. 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 [3], for instance, has relatively small nonzero fill-in during factorization while the model PILOT suffers a large amount of fill-in (see Figure2).

Model	Matrix A			Matrix AA^T	Cholesky factor	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%
RAT1	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

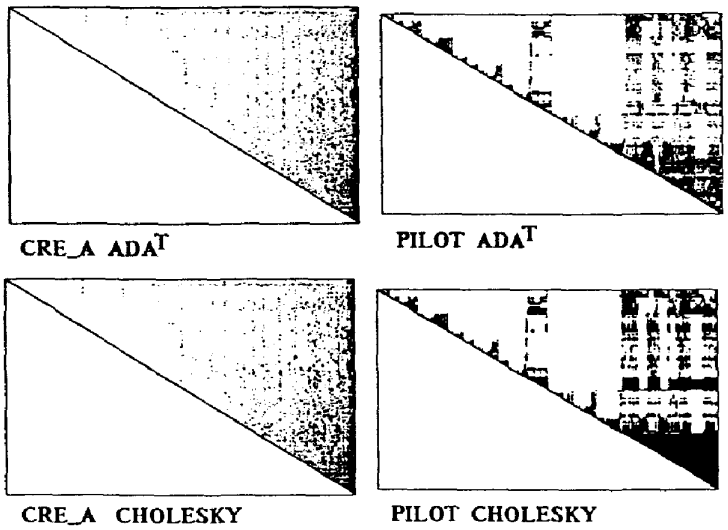

Figure 2: Symmetric matrix and Cholesky factor for the models CRE_A and PILOT

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. There are implementations of the IPM that take advantage of shared memory [17], distributed memory [l], and vector computers [10].

In many cases, the matrices that benefit the least in the parallel solvers are those that are easily solved in the naive sequential way. For very sparse matrices with evenly distributed nonzeros and limited fill-in (e.g. the LP problem CRE_A) the overhead in utilizing parallel factorization is almost always greater than the benefits. These types of problems can be recognized in advance and solved by using a serial implementation. Our main target is to speed up the solution process for those problems that cannot be solved in reasonable time on standard sequential computers.

3 The MasPar computer

Parallel computers differ widely in their design, but there are abstract models that can help in classifying them [8]. One such model is the *SIMD,* which stands for Single Instruction stream, Multiple Data stream. As the name indicates there is only one program for all the processors, but each processor uses a different data set. Another

implied feature of SIMD computers is synchronous operation, that is, all instructions are performed in lock step on all the processors.

3.1 The MasPar MP-2 machine

The MasPar MP-2 system is a massively parallel SIMD computer. It is an upgrade of the older MP-1 system [2], with more powerful processor elements but uses the same communication subsystem. The MP-2 consists of two units: a work station, which acts as a front-end for the system, and a parallel unit. The parallel unit contains between 32 x 32 (1K) and 128 x 128 (16K) processor elements. These are arranged in a 2-dimensional, toroidal-wrapped grid called the processor array. The parallel unit also contains an array control unit, which provides an interface between the front-end and the processor elements.

Following the SIMD paradigm, all the processor elements of the MP-2 receive the same instructions from the control unit at the same time and execute them on their local data. However, individual processor elements can be disabled based on logical expressions and can use indirect references when referring to local data. The advantage of indirect references is that even though all the processors are accessing the *i*'th element of a local array at the same time, *i* can be different on different processors, thus allowing greater flexibility in programming.

The MP-2 provides two types of communication between the processor elements called *Xnet* and *Router*. Xnet communication is the faster, but more restricted procedure. It follows the grid lines of the processor array. Processor elements can send data any distance to the north, south, west, and east, as well as to the northwest, northeast, southwest, and southeast (see Figure 3). The grid lines wrap around, so each processor element always has a neighbor in each of these eight directions.

The basic Xnet communication time is determined by the formula

$$<startup> + <\#bits> * <dist>,$$

where <*startup*> is the latency startup time, <*#bits*> is the number of bits to be sent, and <*dist*> is the distance expressed in number of processors. A typical execution time for 64-bit operands is 6 + *66**<*dist*> clock cycles, with <*dist*> at most 128. On the MP-2 a clock cycle is 80 ns. The MasPar also provides pipelined variations of Xnet that are faster for long distances communication on the grid. They can be used to efficiently broadcast values along rows or columns of the processor array. Sending a 64-bit data

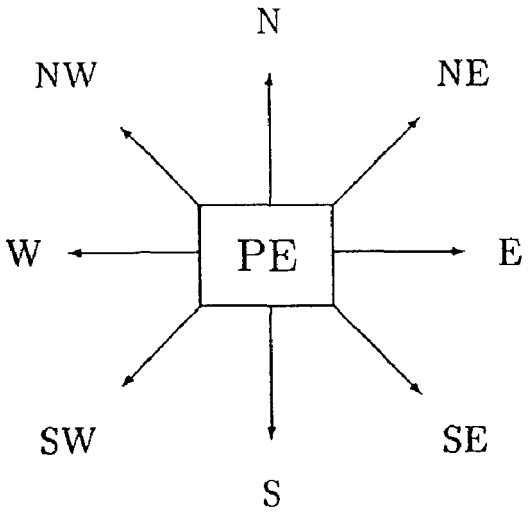

Figure 3: A processor element and its grid connections

item to the other 127 processors in the same row/column of a 128 x 128 machine takes only about three times as much time as a basic Xnet to a nearest neighbor.

Router communication allows each processor to send data to any other processor in the processor array. This makes it more flexible than the Xnet, but slower. The time for a Router communication varies with the amount of collisions, but averages out to about 6200 clock cycles for 64-bit operands.

In the programs described in this paper we use the pipelined Xnet almost exclusively for communication between processor elements. This gives higher speed, but requires that data be distributed to the processors in a special way in order to take advantage of the grid communication.

Each processor element of the MP-2 is a 32-bit load/store arithmetic processor with 40 32-bit registers and 64Kb of RAM. There is no floating point hardware, so all floating point operations are implemented in software. If we define the average time of a floating point operation (flop) as $a = \frac{1}{2}\ (Mult + Add)$, the peak speed of a single processor element is 0.1412 Mflop/sec for 64-bit arithmetic. A 16K processor machine thus has the peak performance of 2314 Mflop/sec.

Comparing the speed of arithmetic to communication on the MP-2, we obtain the ratio

$$\frac{X\,net[1]}{\alpha} = 0.8.$$

Thus floating point arithmetic on a 64-bit value is actually *more* expensive (by 20%) than sending that value to the nearest neighbor in the processor array. Copying a 64-bit value to all the other processors in a column or a row of the processor array, using XnetC, costs only 2.5 times more than a single 64-bit floating point operation on the

MP-2.

This favorable ratio of communication to arithmetic speeds is probably due to the fact that the processors do all floating point computations in software, since the absolute communication speeds are not unusually high.

4 Computing kernels for the MasPar

In this section we consider the adaption of the predictor corrector IPM algorithm to the MasPar MP-2 and discuss some of the design issues, especially our choice of computing kernels for this SIMD parallel computer.

An existing implementation by Levkovitz [9] of the predictor corrector IPM for sequential computers was used as a basis for the parallel implementation. In section 2 we identified a number of computationally intensive steps in the sequential implementation that could be efficiently adapted to a parallel computer. We have rewritten them to execute on the parallel unit of the MasPar. Thus, the basic framework of the sequential implementation remains on the front-end computer of the MasPar, performing preprocessing, loop control, and minor computational tasks, while the bulk of the computation takes place on the parallel unit.

Since the speed of data movement between the front-end and the parallel unit is not very high, it is important that the division of work between the two units does not result in large amounts of data traffic. In our implementation, this is achieved by keeping the constraint matrix A on both the front-end and the parallel unit. This allows us to confine the data transfer in each iteration to sending dense vectors, avoiding the expensive transmission of matrices.

The most obvious candidate for parallelization in the iterative phase of the algorithm is the solution of the two linear systems (see equations (9) and (10))

$$\Delta_P y \;=\; H^{-1}[d_p - AD(z + d_D)] \tag{12}$$

$$\Delta_c y \;=\; -H^{-1}[X^{-1}(\mu e - \Delta_p X \Delta_P Z e)] \tag{13}$$

where $H = ADA^T$. As shown in section 2 these solutions are the main steps in the calculations of the predicting and the correcting projections respectively.

The linear systems can be solved either by a direct method or an iterative one. Although iterative methods are usually better suited to SIMD parallelism we decided to use the direct method of Cholesky factorization. The reason for this choice is the

better numerical stability of the Cholesky factorization and the fact that we are using the predictor corrector IPM, which requires the solution of the same system twice with different right hand sides. Using an iterative method we would have to solve the two systems completely independently, while we only have to compute the Cholesky factorization once and then carry out a pair of triangular solves for each right hand side.

The solution of the linear systems proceeds by first calculating the Cholesky factorization of the m x *m* matrix $H = ADA^T$, where A is the original m x *n* constraint matrix and *D* is a diagonal matrix that changes in every iteration. Then, the two linear systems can be solved by the solution of triangular systems involving the Cholesky factor matrix *L*. Let $r = dp - AD\,(z + d_D)$ be the right hand side of equation (12), then we can compute $\Delta_p y$ by first using forward elimination to solve the triangular system $Lw = r$ and then applying back substitution to solve $L^T \Delta_p y = w$. Similarly $\Delta_c y$ is computed from *L* and the right hand side $t = X^{-1}(\mu e - \Delta_p X \Delta_p Z e)$ of equation (13).

Thus, there are three tasks that require the lion share of the work in an IPM iteration. The first one is to build the matrix *H*, given the diagonal matrix *D* for the current iteration. In a typical sequential implementation this step takes about 10-20% of the time for an iteration (see Table 2 in section 2). The second expensive task in an iteration is the Cholesky factorization of *H* into LL^T. This typically consumes 60-80% of the sequential iteration time. The third step consists of the triangular solutions needed to solve the two linear systems using the Cholesky factor *L*. As triangular solutions require an order of magnitude fewer arithmetic operations than the two other tasks $(0(n^2)$ as opposed to $0(n^3)$ in the dense case) they take up a smaller percentage of the sequential iteration time, usually less than 5%.

In the following subsections we describe in more detail how each of these tasks is performed on the parallel unit of the MasPar MP-2.

4.1 Building ADA^T

The constraint matrix *A* is loaded onto the processor grid in the preprocessing phase of the IPM computation. The columns of *A* are wrapped onto the processor columns of the grid, so that processor column *p* contains the columns *j* with $(j \bmod 128) = p$. For the rows of *A* we use the same layout scheme as for the columns and rows of *L*. This scheme is described in Manne and Hafsteinsson [14].

Thus each row of *A* is stored along a single processor row of the MasPar, but a processor row can contain many matrix rows.

The computation of ADA^T starts by calculating *AD,* storing it in the same way as *A* is stored, and then computing the matrix product (AD) x A^T. Initially we need to send the diagonal of the matrix *D,* stored in the vector *d,* from the front-end to the parallel unit. MasPar provides a communication primitive, called *Blackout,* for copying a matrix from the front-end to the processor grid. If the matrix is 128 x 128 then each processor on the processor grid receives the element in the corresponding position in the matrix. If the matrix is smaller, then the elements of the matrix are sent to a designated subgrid of the processors.

We want each processor row to have a copy of d, so the computation of *AD* can be performed simultaneously on all the processor rows. One way of doing this would be to load a 128 x 128 matrix *S* on the front-end with the first 128 elements of *d* in each of its rows. This would require [*n*/128] Blockout operations to send all the elements of d.

A faster method for sending *d* to the parallel unit is to pack it into the matrix *S* by rows, so that S_{ij} would get element $d_{(i-1)*128+j.}$ Then, after performing a Blockout, each processor row sends copies of the elements they received to all the other rows, using the grid communication primitive Xnet. Since Xnet is local to the processor grid, it is several orders of magnitude faster than Blockout. Using this method only one Blockout transmission is required for every 16K (128 * 128) elements of *d.*

After *d* is distributed, each processor multiplies its part of *A* with the corresponding elements of d, and stores the resulting elements of *AD* in the same way as the *A* elements are stored.

When computing the matrix multiplication *(AD)* x A^T, we take advantage of the fact that we know the structure of the resulting matrix. Thus, we only need to perform the dot product of row *i* of *AD* with column *j* of A^T when element (i, j) of *H* is nonzero.

The algorithm computes one column of *H* at a time. When calculating column *j* it sends the nonzeros of column *j* of A^T, which is stored as row *j* of A, to all the other processor rows. The processors then store it in their local array *guest.* In order to determine which elements of *guest* are valid in each iteration the corresponding element of integer array *time_stamp* is set to *j,* the current iteration number.

For each nonzero l_{ij} of column *j* of *H,* if it is a nonfill element then its row position *i* is sent to all the processor in the same processor row. The processors then proceed to form their local dot product of row *i* of *AD* and column *j* of A^T. These dot products are collected up along the processor row and stored as the value of l_{ij}. With a 128 x 128 processor grid we can in this way compute up to 128 nonzeros of *H* simultaneously.

The complete algorithm for computing ADA^T is given in Figure 4. In the algorithm

```
for each nonzero a,j on this processor do
    c_ij : = a_ij * d_ji
end-do

Alow  : = 1;
Llow  : = 1;
for J : = 1 to M do
    if Arow_name[Alow] = J
      for each nonzero a_J, t do
          copyS[P].di := a_J, t;
          copyS[P]. k : = t;
          all
              time_stamp[k] : = J;
              guest [k] : = fv,
          end-all
      end-do
      Alow := AlOW + 1;
    end-if

    if Lcol_name[Llow] = J
      for each nonfill nonzero l_t,J do
          copyE[P].i := t;
          all
              dot := 0.0;
              for each nonzero c_i,k  in row i of AD do
                  if time_stamp[k] = J
                    dot := dot + c_i,k * guest[k];
                  end-if
              end-do
          end-all
          l_t,J := ScanAdd(dot);
      end-do
       Llow := Llow + 1;
    end-if
end-do
```

Figure 4: The computation of ADA^T

capital variables are global and have the same value for all processors. The all statement makes all the processors active and *copyS[P]*. *y = x* is the version of Xnet that copies the value of variable *x* into the variable *y* on the next *P* processors to the south. Since the processor grid wraps around we use this statement to broadcast the value of *x* on one processor into *y* on all the processors in the same processor column. The construct **Scan Add** is used to add up values along a processor row.

There are other approaches to building the matrix *H*. One that is quite often used is to precompute elementary products.

$$h_{jj} = \sum_{k=1}^{n} a_{ik}\, d_k a_{jk} = \sum_{k=1}^{n} (a_{ik} a_{jk}) d_k .$$

Since only d_k changes in each iteration we can precompute the products $a_{ik}a_{jk}$, for k = l,...,n, and store this list with h_{ij} In this way there is no need to send the elements of *A* between the processors during the iteration and the computation at each step can also be cut down. However, for this method there can be up to n elementary products associated with each h_{ij}. Therefore, we decided not to take this approach, since the additional memory requirement would have severely limited the size of solvable problems.

4.2 Cholesky factorization

Cholesky factorization takes as input an *m* x *m* symmetric matrix *H* and produces the lower triangular matrix L, such that $H = LL^T$. Cholesky factorization is often described in terms of the column operation *cmod* and *cdiv*. The operation cdiv(*j*) is

$$l_{kj} = \frac{h_{kj}}{\sqrt{h_{jj}}}, \qquad k = j, \ldots, m$$

and cmod(*i*, *j)* is

$$h_{ki} = h_{ki} - l_{ij} * l_{kj}, \qquad k = i, \ldots, m$$

where $i > j$. Using these two operation the sequential factorization algorithm for a sparse matrix *H* is given in Figure 5.

The parallel Cholesky factorization algorithm that we use is from Manne and Hafsteinsson [14] and is a parallel version of the algorithm in Figure 5. It maps the nonzero elements of *H* onto the processor grid in a certain way and then computes one column of *L* at a time by parallelizing the operations cmod and cdiv.

```
for j = 1 to m do
    cdiv(j);
    for i > j where h_ij ≠ 0 do
        cmod(i, j);
end-do
```

Figure 5: Sparse Cholesky factorization

To maximize parallelism we try to do as many cmod operations as possible simultaneously. However, if two columns i and $I^{'}$ of H are mapped to the same processor column then cmod(i, j) and cmod(i', j) have to be done sequentially. Because of the SIMD nature of MasPar all the other processor will have to wait for this computation (if they do not have two cmods to perform themselves). Thus, it would be better to assign matrix columns i and i' to different processor columns. (Of course, this might cause two other cmods operations to collide). We can not avoid these collisions entirely, since the number of matrix columns is generally much larger than the number of processor columns.

In Manne and Hafsteinsson [14] this assignment problem is modeled as a graph coloring problem and an approximation algorithm is developed for it. In the current IPM implementation this approximation is done in the preprocessing phase and the layout that it produces is used to map both the columns and the rows of ADA^T onto the processor columns and rows of the MasPar.

4.3 Triangular solutions

When starting the triangular solutions we assume that the Cholesky factor L already resides on the processor grid. For the forward solve $Lw = r$ the parallel unit needs the right-hand-side vector r from the front-end. This vector resides on the diagonal processors of the grid using the same mapping as the factor matrix L. We use a similar technique for sending the vector r as was used for sending the diagonal of D, so that potentially up to 16K elements of r are sent simultaneously.

The parallel forward elimination algorithm computes one variable of the vector w at a time, starting with w_1. This value is sent down the processor column containing column L_{*1} so that it can be multiplied with the elements l_{k1} that are nonzero. The processors containing column L_{*1} do these multiplications simultaneously and then send the result to the diagonal processors in the same row where it is subtracted from the

right hand side. Thus, the diagonal processor containing element r_k sets $r_k = r_k - l_{k1}\, w_1$. The rest of the elements of r are then computed in the same way.

The potential parallelism in the triangular solution algorithm is less than in the Cholesky factorization or in the building of ADA^T. Therefore, in our implementation only up to 128 processors are doing useful work at any point in time. This lack of parallelism is a well known problem with parallel triangular solution. In our case an important issue is that by computing the triangular solutions on the processor grid we avoid having to send the matrix L between the parallel unit and the front-end.

The back substitution $L^T \Delta_p y = w$ is performed in a similar manner to the forward elimination, except that now we start by computing the m'th element of $\Delta_p y$ and work our way up the vector. Back substitution, however, usually takes about 20% longer than forward elimination. The reason for this difference is that the layout of L on the processor grid is optimized for the Cholesky factorization. This layout turns out to be good for forward elimination, but is not as beneficial for back substitution.

Another factor that influences the time for back substitution is the generally higher density of individual rows of L than of the columns. Because of fill-in, L usually has a rather dense block of nonzeros in its bottom part. This means that the row with the highest number of nonzeros in L usually has many more nonzeros than the column with the highest number of nonzeros. Row L_{1*} only has one nonzero, as has column L_{*m}. Thus, there is usually more variation in the nonzero count of the rows of L than there is in the columns. This makes the back substitution algorithm less efficient, since it needs to access the columns of L^T, which are the rows of L.

5 Experimental results

To test the parallel SIMD implementation of the IPM algorithm, we used the models described in section 2. The different density and size of the models are useful in finding the threshold where utilizing the MasPar becomes advantageous. The model statistics and the number of IPM iterations it takes to solve them on a SUN Spare 10 computer are listed in Table 1.

Tables 3 and 4 show timing results for a single iteration of the IPM on the MasPar. Table 3 gives the absolute time in seconds for the various tasks and Table 4 displays the percentages. The column "Data trans." contains the time spent in data transmission between the front-end computer and the parallel grid. The columns “Build ADA^T" and "Cholesky[1]' contain the amount of time for building the symmetric matrix and factoring

Model	Data trans.	Build ADA^T	Cholesky	Triag. Solves	Other	Total Time
CAR2	0.197	1.084	0.195	0.590	0.809	2.875
25FV47	0.187	1.402	0.262	1.058	0.427	3.336
PILOT	0.199	3.477	0.894	2.629	1.539	8.738
CAR11	0.247	21.367	5.656	9.144	14.816	51.230
BNL2	0.269	3.773	0.848	2.938	0.981	8.809
RAT1	0.309	9.621	3.848	9.070	2.871	25.719
CRE_A	0.542	5.555	1.074	3.980	1.966	13.117
DFL001	0.477	11.277	8.516	13.820	2.515	36.605
CAR4	1.171	28.969	2.223	8.504	5.512	46.379
CAR8	—	—	39.371	55.272	—	—

Table 3: Timings in seconds for one iteration of the MasPar IPM

it respectively. The time required for the four triangular solves that are performed in each iteration is given in the column "Triag. Solves". The column "Other" gives the time that the front-end computer requires to perform the remaining computations in each iteration. These include two multiplications of the matrix A by a dense n-vector and various vector computations.

There was not enough memory available on the front-end to solve the model CAR8. However, we present some figures for CARS in Table 3 showing that the parallel unit has the potential to solve larger models than the ones we managed to compute.

The results obtained on the MasPar are compared to the results obtained from a sequential implementation of the IPM running on a SUN Spare 10 model 25 workstation. From the three different implementations of the Cholesky factorization algorithm [10], we report only the best times for each model. These results are given in Table 5.

Let us now consider when we should use the MasPar implementation rather than the sequential Spare 10 implementation. To have a better basis for comparing the two implementations we need to consider if there are some simple parameters of the model that influence each task in the iteration.

In the sequential implementation the time required to build the matrix ADA^T is usually proportional to the number of nonzeros in ADA^T. The execution time for the other tasks in the iteration is mainly related to the number of nonzeros in the Cholesky factor matrix L.

The time it takes to build ADA^T in the MasPar implementation depends primarily on the number of rows in A and the number of nonzeros in ADA^T. The execution time

Model	Data trans.	Build ADA^T	Cholesky	Triag. solves	Other
CAR2	6.9%	37.7%	6.8%	20.5%	28.1%
25FV47	5.6%	42.0%	7.9%	31.7%	12.8%
PILOT	2.3%	39.8%	10.2%	30.1%	17.6%
CAR11	0.5%	41.7%	11.0%	17.9%	28.9%
BNL2	3.1%	42.8%	9.6%	33.4%	11.1%
RAT1	1.2%	37.4%	15.0%	35.3%	11.1%
CRE_A	4.1%	42.3%	8.2%	30.4%	15.0%
DFL001	1.3%	30.8%	23.3%	37.7%	6.9%
CAR4	2.5%	62.5%	4.8%	18.3%	11.9%

Table 4: Percentage of time for different tasks in the MasPar IPM

Model	Build ADA^T	Cholesky	Traig. Solves	Other	Total time
CAR2	2.288	3.112	0.105	1.296	6.801
25FV47	0.126	0.498	0.051	0.293	0.968
PILOT	1.431	6.814	0.365	2.310	10.920
CAR11	339.000	1232.230	12.166	302.764	1886.160
BNL2	0.121	1.992	0.154	0.735	3.002
RAT1	2.598	79.954	2.165	3.634	88.351
CRE_A	0.464	0.402	0.076	0.694	1.636
DFL001	0.606	180.773	2.691	5.479	189.549
CAR4	4.395	15.397	0.324	7.204	27.320
CAR8	276.950	1725.450	12.640	263.971	2279.011

Table 5: Timings in seconds for one for iteration of the Sparc 10 IPM

of the parallel Cholesky factorization can be linked to the number of nonzeros in L. The time for the triangular solves also depends on the number of nonzeros in L, but is in addition influenced by the number of rows in A.

Thus, there are three simple parameters that we can use a priori to predict whether a model will be solved faster on a MasPar than a sequential computer. These parameters can be summarized as

(i) the number of rows in A
(ii) the number of nonzeros in ADA^T
(iii) the number of nonzeros in the factor matrix L

The larger the number of nonzeros in ADA^T and L, the more efficient the MasPar implementation is. On the other hand, as the average number of nonzeros per column decreases the less effective the MasPar becomes. We observe that for small and sparse models (e.g. 25FV47) the sequential implementation is faster. As the problems grow in size the massively parallel implementation becomes more efficient.

Based on the above observations, we can conclude that there exists an breakeven point, above which it is worth while using a massively parallel SIMD computer to solve the computationally intensive parts of the IPM. For example, in our implementations the size and density of the model PILOT seems to lie near this point.

For a simple rule of thumb to establish when the MasPar is faster than the Spare 10 we can combine two of the above parameters and consider the average number of nonzeros per column in L. In most of our parallel algorithms we have each processor column of the MasPar working on one matrix column at a time. Since there are 128 processors per processor column it is important that there is enough work in each matrix column to keep most of the processors busy performing useful calculations.

In Table 6 we have calculated this parameter for each of the models and we also indicate which implementation is faster at solving the model. Using Table 6 we can estimate that when the average number of nonzeros per column in L is above 100 then the MasPar implementation will be faster than the sequential Sparc 10 implementation. For another, faster sequential computer this figure would be higher, or, if the sequential computer was fast enough, the 128 x 128 processor MasPar might never be able to reach the same speed no matter how high the average number of nonzeros per column in L.

A more accurate formula for the breakeven point could be found by calculating the amount of computation required in each of the tasks in the IPM iteration for the model in question. However, we feel that the cost of calculating those quantities defeats

Model	Ave. # nonz. Per col in L	Suitability of MasPar(M) or Sparc(S)
CAR2	153.5	M
25FV47	40.7	S
PILOT	142.6	M
CAR11	765.7	M
BNL2	39.3	S
RAT1	399.1	M
CRE_A	10.5	S
DFL001	258.2	M
CAR4	10.4	S
CAR8	191.7	—

Table 6: The average number of nonzeros per column in L

our purpose. The above simple rule of thumb, quickly and with reasonable accuracy, indicates whether it is worth solving a particular model on the MasPar.

6 Conclusions

In this paper we have presented an implementation of the interior point method for a massively parallel SIMD machine. We investigated its performance on various standard industrial test problems and tried to determine the types of problems it is best suited for.

An obvious conclusion from our work is that a massively parallel SIMD computer does not process all problems equally effectively. The problems need to be of some minimum size for it to be worth solving them on such a machine. Since each individual processor of the MasPar is not very powerful we need to have most of them contributing to the solution at all times. If the problem is too small it is difficult to achieve any performance advantage.

A related consideration is the division of computational load between the front-end computer and the parallel unit of the MasPar. In a large application like the interior point method there are always tasks that do not parallelize well. If only a handful of the parallel processors can be applied to the solution of a task, then it runs faster on the front-end than on the parallel unit. Thus, it is important to correctly identify those parts of an application that can be translated into efficient data parallel algorithms.

It is worth investigating if our IPM implementation will benefit from using an iterative method, e.g. the conjugate gradient algorithm, instead of Cholesky factorization to solve the linear systems in each iteration of the IPM. Parallel SIMD computers are often considered better suited for iterative algorithms than direct ones, since iterative algorithms are usually structurally simpler. On the other hand, in our implementation the Cholesky factorization and the triangular solutions usually take less than 50% of the time in each iteration, so there is less room for improvement than there is in the sequential case.

Acknowledgment

We would like to thank Para//ab, the Parallel Processing Laboratory at the University of Bergen for the use of their MasPar computer. We would 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.

References

[1] 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).

[2] T. BLANK, *The MasPar MP-1 architecture,* in Proceedings of IEEE Compcon Spring 1990, IEEE, February 1990.

[3] 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.

[4] I. DUFF, A. ERISMAN, AND J. REID, *Direct Methods for Sparse Matrices,* Oxford University Press, 1986.

[5] 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.

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

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

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

[9] 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.

[10] —, *Solving large scale linear programming problems using a interior point method on a vector computer,* Tech. Report TR/06/93, Brunel, The University of West London, August 1993.

[11] 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.

[12] —, *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.

[13] 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.*

[14] F. MANNE AND H. HAFSTEINSSON, *Efficient sparse Cholesky factorization on a parallel SIMD computer,* Tech. Report CS-93-84, Department of Informatics, University of Bergen, 1993.

[15] 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.

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

[17] 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.

[18] 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.

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