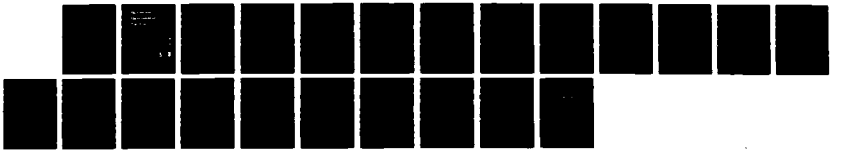
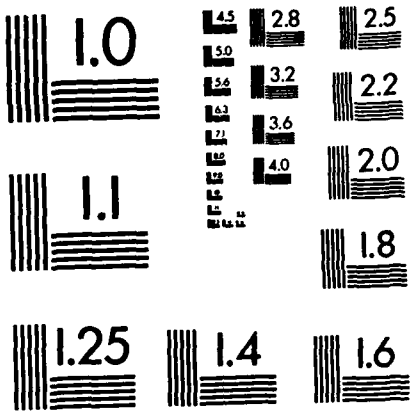
AD-A193 137 A REVIEW OF PARALLEL METHODS FOR SOLVING SETS OF LINEAR 1/1
EQUATIONS AND THE..(U) HATFIELD POLYTECHNIC (ENGLAND)
NUMERICAL OPTIMISATION CENTRE.. L C DIXON NOV 87
UNCLASSIFIED TR-188 DAJA45-87-C-0038 F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1

Numerical Optimisation Centre

TECHNICAL REPORT NO. 188

NOVEMBER 1987

A REVIEW OF PARALLEL METHODS FOR SOLVING SETS OF LINEAR EQUATIONS AND THEIR APPLICATION WITHIN OPTIMISATION ALGORITHMS

by

L C W Dixon

Hatfield Polytechnic

AD 1987

OPTIMISATION ALGORITHMS FOR HIGHLY PARALLEL COMPUTER ARCHITECTURES

First Interim Report

"A REVIEW OF PARALLEL METHODS FOR SOLVING SETS OF LINEAR EQUATIONS AND THEIR APPLICATION WITHIN OPTIMISATION"

by

L.C.W. DIXON

November 1987

United States Army

EUROPEAN RESEARCH OFFICE OF THE U.S. ARMY

London England

CONTRACT NUMBER DAJA45 - 87 - C - 0038

The Hatfield Polytechnic

THE HATFIELD POLYTECHNIC

NUMERICAL OPTIMISATION CENTRE

A REVIEW OF PARALLEL METHODS FOR SOLVING SETS OF LINEAR EQUATIONS AND THEIR APPLICATION WITHIN OPTIMISATION ALGORITHMS

by

L C W Dixon

TECHNICAL REPORT NO. 188 NOVEMBER 1987

Abstract

When solving optimisation problems on a parallel computing system, the first consideration must be to utilise the parallelism to speed up the 95% of the time typically spent in function and gradient calculations. This is normally done in one of two ways namely the parallel computation of functions or gradient evaluations or the division of each function evaluation into a number of parallel tasks.

Assuming this prime task has been undertaken effectively then for efficiency the other 5% of the computation must also utilise the parallelism available on the system.

The dominant remaining calculation is usually the solution of a set of linear equations.

In this paper the implication of parallel processing on the solution method for solving linear equations will be reviewed.

1. Introduction

In this report we will be concerned with the optimisation problem

Min $F(x)$ $\quad x \in R^n$.

Occasionally we will assume simple upper and lower bounds of the form $l_i \leq x_i \leq u_i$ exist.

There is of course a complete theory for the convergence of iterative algorithms which generate a sequence of estimates

$$x^{(k+1)} = x^{(k)} + \alpha p^{(k)}$$

and many efficient codes exist for the solution of such problems on a sequential machine. These have successfully solved many optimisation problems.

The question therefore arises as to why then we should be interested in introducing the parallel processing concept into numerical optimisation.

The main reasons that influenced us were:

(1) that we knew of industrial problems that took an embarassingly long time on a sequential machine

and we knew too that

(2) industry only poses problems that it thinks might be soluble.

By introducing the parallel processing concept into numerical optimisation we hoped to be able to extend the range of soluble problems. We identified four different situations where we felt that the solution of optimisation problems would most benefit from the availability of parallel processing machines. These were:-

1) Small Dimensional Expensive Problems

These are typified by industrial problems which frequently have a small dimension $n < 100$ but where the time required to compute the function and gradient values at $x^{(k)}$ can be considerable and where this dominates the computation within the algorithm.

2) Large Dimensional Problems

There are many large dimensional problems n > 2000 where the combined processing time and storage requirements cause difficulties.

3) On Line Optimisation

There are many on line optimisation problems, for instance the optimisation of car fuel consumption which cannot be easily solved using existing sequential optimisation codes on the type of processors that could be easily installed within a car but which might be solved on more than one such processor.

4) Multi Extremal (Global) Optimisation

Problems in which the objective function has many local minima and where the real problem is to identify the best of these, still present many difficulties because the available sequential codes are weak and expensive in computer time.

In all four of these areas the availability of parallel processors promised significant improvements and in each that promise has been achieved. In this lecture we will mainly be concerned with the first class of problem but the solution of sets of linear equations is an essential step in the solution of all four classes of problem.

2. Optimisation Problems and Algorithms

It is usual to find on analysing the solution of most industrial optimisation problems that at least 95% of the computer time is spent in evaluating the values of the objective function $F(x)$ at $x^{(k)}$ and only 5% of the time within the optimisation code.

It is therefore natural to concentrate on the speed up of the calculation of the engineering model $F(x)$ rather than the code.

This can effectively be done in two distinct ways. If we assume a computer system with P processors that can act in parallel.

Approach A

The calculation of each objective function value F(x) is divided into P parallel tasks. This approach leaves the responsibility for the efficient use of parallelism in the hands of the user.

Approach B

The algorithm is modified so that it can accept P values of F(x) or $\nabla F(x)$ computed simultaneously. This places the responsibility for the use of parallelism on the algorithm designer.

At the NOC we have developed codes of both type A and B. The finite element optimisation approach to the solution of nonlinear partial differential equations Singh (1983)[1] naturally leads to optimisation problems that divide into parallel tasks. This has been implemented with very significant speed up on the ICL-DAP by Ducksbury (1985)[2]. Indeed any partially separable objective function Toint (1986)[3] can be solved in this way.

Early implementations of a B type approach in which gradient and Hessian information were approximated by difference in function values at different points computed in parallel were reported by Patel (1984)[4].

With the availability of automatic differentiation Rall (1981)[5] which can be implemented effectively in ADA Mohseninia (1987)[6], codes are now being written to utilise concurrent tasking to calculate the components of the gradient (and when necessary the Hessian matrix) on parallel processors. In this context Mohseninia has found that it is natural and efficient to utilise the partially separable structure when computing the function and gradient values as parallel (concurrent) tasks.

More details of these approaches is given in Dixon & Price (1986)[7].

If we assume that when solving class 1 optimisation problems that typically 95% of the computation is spent calculating function and gradient values then a simple calculation demonstrates that to obtain significant

speed up factors the remaining 5% of the codes must also utilise the parallelism available. Assuming perfect efficiency in calculating the function values in parallel on P processors and letting $\tau(P)$ be the time of computation we would have

$$\tau(P) = \left(\frac{\cdot 95}{P} + \cdot 05 \right) \tau(1)$$

so the ratio $\frac{\tau(1)}{\tau(P)}$ has an upper bound of 20 even if there are 4000 processors.

It is therefore essential to reduce 5% of the time spent in the optimisation code. In most optimisation codes this calculation is dominated by the time required to solve a set of equations

$$Ax = b \qquad x \in R^n$$

typically in unconstrained optimisation this will be some approximation of the Newton equation

$$\nabla^2 Fx = - \nabla F$$

while for REQP methods of constrained optimisation A and b also contain constraint information.

Even for unconstrained optimisation the most apropriate approximations A and b are dependent on both n and P. Byrd, Schnabel and Shultz (1987)[8] discuss modifications to Approach B appropriate when $n + 1 < P < (n^2 + 5n + 2)/2$. Patel's experience was obtained with $P > (n^2 + 5n + 2)/2$; whilst Ducksbury assumed $n + 1 > P$. However in this lecture we will now assume A and b have been obtained appropriately and consider the solution of the set of equations

$$Ax = b.$$

3. The Solution of a Set of Linear Equations: Introductory Comments

The problem of solving a set of equations

$Ax = b$

is one of the most frequently occurring problems in numerical computation. Because of the frequency with which it occurs it has been intensely researched and there are very many variants of most algorithms designed to solve it. It is probably however true that on a single processor machine most numerical analysts would prefer to use one of four standard methods if, as will be assumed in this lecture, A is dense.

The four broad classes of method we will consider are:

1. Gaussian Elimination
2. QU Decomposition
3. Iteration
4. Conjugate Direction Techniques.

Within the first category we will consider four variations:

1.1 Choleski Decomposition

1.2 Total Pivoting

1.3 Partial Pivoting

1.4 Gauss-Jordan.

Similarly within the second category we have:

2.1 Householder's Method

2.2 Givens' Method

2.3 The Gramm Schmidt Method

and in the third category:

3.1 S.O.R.

3.2 Gauss-Seidel Method

3.3 Jacobi Method.

In each of the above three classes of method, I have attempted to list the variations in the popular order for a sequential machine, i.e. on a

sequential machine I suspect most people would use Choleski decomposition if the matrix is symmetric and positive definite. Whilst on a nonsymmetric or nonpositive matrix most people prefer to use a partial pivoting code, the case for total pivoting on really badly conditioned problems is accepted but disliked. The Gauss-Jordan variant of either is rejected as more expensive.

The questions we shall consider in this lecture are which algorithm we should use in R^N with p processors and secondly how does that answer vary with n and p and the relative compute and data transfer times of the system.

4. Gaussian Elimination Methods

Most Gaussian codes consist of three parts

(1) triangularisation

(2) pivoting

(3) back substitution

in which the triangulation and pivoting interact and are both completed before the back substitution starts.

4.1 Triangularisation

Without pivoting triangularisation takes the simple form

```
FOR k = 1 TO N - 1
FOR i = k + 1 TO N
FOR j = k + 1 TO N                                    F1
a_ij = a_ij - a_ik a_kj / a_kk
NEXT j, i, k.
```

Most early numerical analysis texts commented that it is preferable to rewrite this as:

```
FOR k = 1 TO N - 1
FOR i = k + 1 TO N
c = a_ik/a_kk                                                    F2
FOR j = k + 1 TO N
  a_ij = a_ij - ca_kj
NEXT j, i, k.
```

This change was recommended because on most sequential machines

t{1 DIV + (N - k) mults} < t{(n - k)Divs}.

Parkinson observed that this is no longer true on either a pipeline or a DAP processor. On a pipeline

t{1 Div + pipeline mult} > t{chained mult/divisions}

whilst on the DAP the original code is 3 simple matrix operations, a property lost in the modified form.

For those using Fortran it is often further recommended that for data access reasons it is preferable to do

```
FOR k = 1 TO N - 1
FOR i = k + 1 TO N
c_i = a_ik/a_kk                                                  F3
NEXT i
FOR j = k + 1 TO N
FOR i = k + 1 TO N
a_ij = a_ij - c_i a_kj
NEXT i, j, k.
```

While these changes may seem trivial such considerations can significantly alter the efficiency of codes and their effect in a particular architecture should be borne in mind.

4.2 The partial pivoting operation

In partial pivoting the largest element of n - k elements in the pivot row is determined between the k and i loops. On a sequential machine this implies n - k comparisons. On a parallel machine the number of comparisons needed to compare R numbers on P machines depends on R and P.

For instance if P = 4, R = 16 it takes 5 steps

Step 1 $C(N_1; N_2)$, $C(N_3, N_4)$, $C(N_5, N_6)$, $C(N_7, N_8)$

2 $C(M_{1,2}, M_{3,4})$, $C(N_9, N_{10})$, $C(M_{5,6}, N_{11})$, $C(N_{7,8}, N_{12})$

3 $C(M_{1,2,3,4}, M_{9,10})$, $C(N_{13}, N_{14})$, $C(M_{5,6,11}, N_{15})$, $C(M_{7,8,12}, N_{16})$

4 $C(M_{1,2,,4,9,10}, M_{13,14})$, - , $C(M_{5,6,11,15}, M_{7,8,12,16})$, -

5 $C(M_{1,2,3,4,9,10,13,14})$, $M(_{5,6,11,15,7,8,12,14})$, - , - , - .

The case usually quoted assumes p = R/2 when the comparison requires $\log_2 R$ steps if $R = 2^k$.

4.3 The partial pivoting algorithms

On a sequential machine the algorithm requires $\frac{1}{3}n^3 + O(n^2)$ multiplications and divisions, and $O(n^2)$ comparisons. The times for the comparisons are usually ignored.

On a parallel machine with $P = (n - 1)^2$ processors Sameh and Kuck (1977) showed the most active processor only needed 3(n - 1) multiplications/ divisions, but O(n log n) comparisons. For large n pivot comparisons dominate! Lord, Kowalik & Kumar (1983) who tested many algorithms on the HEP (P = 8) computer for which the assumption $P = (n - 1)^2$ was inappropriate, re-examined the algorithm assuming $P = \lfloor n/2 \rfloor$, they showed that the algorithm then requires $n^2 - 1$ multiplications and divisions on the critical path.

Their Speed up = $\frac{1}{3}n^3/n^2 - 1 = \frac{1}{3}n$ Efficiency = $\frac{1}{3}n/\frac{n}{2} = \frac{2}{3}$.

The Sameh/Kuck and Lord/Kowalick & Kumar methods are contrasted below for n = 4.

Sameh/Kuck P = 9

1. Select Pivot and form $r_k = 1/a_{kk}$
2. Calculate $c_i = a_{ik} r_k$ (PAR)
3. $a_{ij}^{(2)} = a_{ij}^{(1)} - c_i a_{kj}^{(1)}$ (PAR)
4. Select pivot and form $r_k = 1/a_{kk}$
5. Calculate $c_i = a_{ik}^{(2)} r_k$ (PAR)
6. $a_{ij}^{(3)} = a_{ij}^{(2)} - c_i a_{kj}^{(2)}$ (PAR)
7. Select pivot and form $r_k = 1/a_{kk}$
8. Calculate $c_i = a_{ik}^{(3)} r_k$
9. $a_{ij}^{(4)} = a_{ij}^{(4)} - c_i (a_{kj}^{(4)}$

Note. L/K/K could be improved at the beginning but this would alter their calculation

Lord Kowalik & Kumar P =2

1. The same
2. $c_2 = a_{2k} r_k$
3. $c_3 = a_{3k} r_k$
4. $c_4 = a_{4k} r_k$
5. $a_{22}^{(2)}$ $a_{23}^{(2)}$
6. $a_{32}^{(2)}$ $a_{33}^{(2)}$
7. $a_{42}^{(2)}$ $a_{43}^{(2)}$
8. Pivot and r_k $a_{24}^{(2)}$
9. c_3 $a_{34}^{(2)}$
10. c_4 $a_{44}^{(3)}$
11. $a_{33}^{(3)}$ $a_{34}^{(3)}$
12. $a_{43}^{(3)}$ $a_{44}^{(3)}$
13. Pivot and r_k
14. c_4
15. $a_{44}^{(4)}$.

Amusing variants can be constructed, for instance P = 3

1. Select pivot and form $r_k = 1/a_{kk}$
2. c_2 c_3 c_4
3. $a_{22}^{(2)}$ $a_{23}^{(2)}$ $a_{24}^{(2)}$
4. $a_{32}^{(2)}$ $a_{33}^{(2)}$ $a_{34}^{(2)}$
5. $a_{42}^{(3)}$ $a_{43}^{(3)}$ $a_{44}^{(3)}$
6. Select pivot and form $r_k = 1/a_{kk}$
7. c_3 c_4
8. $a_{33}^{(3)}$ $a_{34}^{(3)}$
9. $a_{43}^{(3)}$ $a_{44}^{(3)}$
10. Select pivot and form $r_k = 1/a_{kk}$
11. c_4
12. $a_{44}^{(4)}$.

Comparing the four codes we have

P = 1	C = 23	C*P = 23
P = 2	C = 15	C*P = 30
P = 3	C = 12	C*P = 36
P = 9	C = 9	C*P = 81

so our speed up is always obtained at a processor * time cost.

No attention has been given so far to data transfer times i.e. the time necessary to transfer the data to the processor undertaking the compute. We will return to this point in section 4.5.

4.4 Total pivoting

In total pivoting at iteration k we need to compare $(n - k)^2$ numbers. This would dominate the cost even more for large n and small P; but is ideal on the ICL DAP where there is a special fast $\text{Max}(A_{ij})$ instruction for determining the maximum element of a matrix.

4.5 The Gauss Jordan method

Gaussian elimination is usually completed by back substitution.

Kuck (1977) has shown that this can be done in 3(n - 1) steps on n - 1 processors. However it should never be done on such a system since the unused processors are available precisely when they are needed to do the Gauss Jordan calculation. If we insert n additional rows to update RHS we now only need one additional step to complete the solution.

For data transfer considerations it is better to use n or (n + 1) processors than n - 1 as then computations involving one row or column can be done on a particular processor. If this is done then it is efficient to update the pivot row in parallel at each step by making the pivot element unity.

Gauss Jordan $\quad n = 4 \quad P = 4$

1. Select pivot and form $r_k = 1/a_{kk}$
2. $c_1 \qquad c_2 \qquad c_3 \qquad c_4$
3. $a_{12}^{(2)} \qquad a_{22}^{(2)} \qquad a_{32}^{(2)} \qquad a_{42}^{(2)}$
4. $a_{13}^{(2)} \qquad a_{23}^{(2)} \qquad a_{33}^{(2)} \qquad a_{43}^{(2)}$
5. $a_{14}^{(2)} \qquad a_{24}^{(2)} \qquad a_{34}^{(2)} \qquad a_{44}^{(2)}$
6. $b_1^{(2)} \qquad b_2^{(2)} \qquad b_3^{(2)} \qquad b_4^{(2)}$
7. Select pivot and form $r_k = 1/a_{kk}$
8. $c_1 \qquad c_2 \qquad c_3 \qquad c_4$
9. $a_{13}^{(3)} \qquad a_{23}^{(3)} \qquad a_{33}^{(3)} \qquad a_{43}^{(3)}$
10. $a_{14}^{(3)} \qquad a_{24}^{(3)} \qquad a_{34}^{(3)} \qquad a_{44}^{(3)}$
11. $b_1^{(3)} \qquad b_2^{(3)} \qquad b_3^{(3)} \qquad b_4^{(3)}$
12. Select pivot and form $r_k = 1/a_{kk}$
13. $c_1 \qquad c_2 \qquad c_3 \qquad c_4$
14. $a_{14}^{(4)} \qquad a_{24}^{(4)} \qquad a_{34}^{(4)} \qquad a_{44}^{(4)}$
15. $b_1^{(4)} \qquad b_2^{(4)} \qquad b_3^{(4)} \qquad b_4^{(4)}$
16. Form $r_k = 1/a_{kk}$
17. $c_1 \qquad c_2 \qquad c_3 \qquad c_4$
18. $b_1^{(5)} = x_1 \quad b_2^{(5)} = x_2 \quad b_3^{(5)} = x_3 \quad b_4^{(5)} = x_4$

It is probably worht repeating that on most parallel machines it will be more efficient to omit the separate calculation of r_k and compute c_i by division rather than multiplication. If the pivot calculation is ignored the number of steps is

$$\sum_{k=1}^{n+1} k - 1 = \tfrac{1}{2}(n+1)(n+2) - 1.$$

5. Orthogonal AU Decomposition Techniques

The parallel analysis of both Householder's and Givens' method was given by Sameh and Kuck (1977). They showed that given $P = O(n^2)$ processors Householder's method required O(n log n) operations but Givens' method only required O(n). This contrasts with the sequential situations where Householder's method is the more efficient. This comparison was tested and confirmed by Sorenson (1985).

The Sequential Givens routine consists of

1. FOR Q = 1 TO N - 1
2. FOR P = Q + 1 TO N

$$S = a_{QQ}/\sqrt{a_{PQ}^2 + a_{QQ}^2} \qquad c = a_{PQ}/\sqrt{a_{PQ}^2 + a_{QQ}^2}$$

3. FOR J = 1 TO N

$$TEMP = C * a_{PJ} + S\ a_{QJ}$$

$$a_{QJ} = -S/* a_{PJ} + C\ a_{QJ}$$

$$a_{PJ} = TEMP$$

4. Next j, P, Q.

Note that the J loop is an ideal pipeline calculation. Typically operation 3 is considered 1 task.

The essential feature of Givens' method is that a pair PQ only alters rows P and Q so a number can be performed in parallel.

The earliest discussion of a parallel Givens code was that due to Sameh and Kuck (77) who reduced A_{PQ} to 0 in the following order for an n x n matrix

$$\begin{bmatrix}
* & & & & & & & \\
1 & * & & & & & & \\
2 & 3 & * & & & & & \\
3 & 4 & 5 & * & & & & \\
4 & 5 & 6 & 7 & * & & & \\
5 & 6 & 7 & 8 & 9 & * & & \\
6 & 7 & 8 & 9 & 10 & 11 & * & \\
7 & 8 & 9 & 10 & 11 & 12 & 13 & *
\end{bmatrix}$$

Sameh & Kuck (77)

$$\begin{bmatrix}
* & & & & & & & \\
3 & * & & & & & & \\
2 & 5 & * & & & & & \\
2 & 4 & 7 & * & & & & \\
1 & 3 & 6 & 8 & * & & & \\
1 & 3 & 5 & 7 & 9 & * & & \\
1 & 2 & 4 & 6 & 8 & 10 & * & \\
1 & 2 & 3 & 5 & 7 & 9 & 11 & *
\end{bmatrix}$$

Modi & Clarke (84)

Their code performed 13 parallel Givens steps on 4 processors. In general (2n - 3) parallel Givens steps on $\frac{n}{2}$ processors. As each parallel Givens step involves 4n + 4 operations the speed up is dominated by $4n^3/3(2n - 3)(4n + 4)$; so $S = \frac{1}{6}n$ and $E = \frac{1}{3}$. Many attempts at improving the Sameh & Kuck method have been proposed but Cosnard (1985) has shown that the "greedy" algorithm, Modi and Clarke (1984), is optimal. This is also shown above, and only requires 11 parallel Givens steps on 4 processors. Cosnard states that the efficiency of this method is asymptotically $\frac{1}{2}$ but if we utilise $\frac{n}{2}$ parallel pipelines when performing the inner loop the algorithm is much more powerful than this would indicate.

Kowalik, Kumar and Kamgria (1983) noted that if 2(k + 1) processors are used to perform the inner loop it only requires 4 steps. Again analysing the nonoptimal sort they concluded that with $P = \frac{3}{4}n^2 + \frac{5}{2}n$ they could obtain a speed up $S = 1n^2/6 + O(n)$, C = 8n - 12. Adding the 3n - 3 operations for parallel back substitution they state the parallel operation count as 11n - 5. They also analysed the case with P = [(n-1)/2] and claimed $C = 6n^2 + 8n - 25$ giving $S = \frac{2}{9}n$ and $E = \frac{4}{9}\frac{n}{n-1}$. These times were verified on their test runs on the HEP.

6. Iterative Methods

6.1 Jacobi's method

To solve Ax = b first scale so $D_{ii} = 1$ then iterate

$$x^{(k+1)} = (I - A)x^{(k)} + b.$$

This method is convergent if $||I - A|| < 1$ but very slow when $||I - A|| > .9$.

The method is however ideal for parallel processing as we can do all the rows in parallel if $P \geq n$ and P rows in parallel if $P < n$. If each processor is a pipeline it is even better, as each row update is an ideal pipeline operation.

It is therefore possible to speed up the very poor method significantly.

6.2 Gauss-Seidel method

The ordinary Gauss-Seidel algorithm

$$x_i^{(k+1)} = x_i^{(k)} - \sum_{j=1}^{i-1} A_{ij}x_j^{(k+1)} - \sum_{j=1}^{n} A_{ij}x_j^{(k)} + b_i$$

is not suitable for parallel computing due to the $x_j^{(k+1)}$ term on RHS.

This can however be adapted for a block Gauss-Seidel approach with $P < n$. If q s.t. $Pq < i < P^{(q+1)}$

$$x_i^{(k+1)} = x_i^{(k)} - \sum_{j=1}^{Pq} A_{ij}x_j^{(k+1)} - \sum_{j=P^{q+1}}^{n} A_{ij}x_j^{(k)} + b_i$$

allows P values of x to be evaluated in parallel.

6.3 Assynchronous Jacobi or Gauss Seidel Method

As an alternative to the Block Gauss Seidel method Baudet (1978)[17] proposed the assynchronous algorithm where

$$x_i^{(k+1)} = x_i^{(k)} - \sum_{j=1}^{n} A_{ij}x_j^{L} + b_i$$

is used where x_j^L is the latest value in the store when it is accessed. Allowing P = n processors, 1 for each i, and slight variations of performance x_j^L could be either $x_j^{(k)}$ or $x_j^{(k+1)}$. His analysis indicated there would be no loss of performance.

7. Conjugate Gradient Method

The sequential code consists of the following steps:

1. Choose $x^{(0)}$ (usually 0); ε; $r_0 = b - Ax_0$, $d_0 = r_0$ while $||r|| > \varepsilon$ do 2,3,4.

2. $\underline{x}_i^+ = \underline{x}_i + \alpha \underline{d}_i \qquad \alpha = r^T d / d^T A d$

3. $\underline{r}_i^+ = \underline{b}_i - A\underline{x}^+$

4. $d^+ = r^+ + \beta d \qquad \beta = R^{+T} r^+ / r^T r.$

Steps 2,3,4 are ideal for a parallel processor.

with P = n, the calculation of Ax and Ad requires n parallel steps

$r^T d$, $d^T Ad$, $r^T r$ requires log n parallel steps.

So one iteration costs $2n + 3 \log n + 4$ steps.

If $p = n^2$ we can do all the multiplications in Ax and Ad in parallel so one iteration is only $3 \log n + 6$ steps.

8. Preconditioning

In practice on a sequential machine both Jacobi and Conjugate Gradients are preconditioned before use. In the Jacobi method this involves the use of either a matrix C or a matrix M where the equation becomes

$$CAx = Cb \quad \text{or} \quad M^{-1}Ax = M^{-1}b.$$

For efficiency we require C to approximate A^{-1}, or M to approximate A; if we are using M we need $Mz = w$ to be much easier to solve than $Ax = b$; while if we are using C we need C to have a nice sparsity pattern.

The preconditioned Jacobi method is then either

$$x^{k+1} = (I - CA)x^k + Cb \quad \text{or} \quad x^{k+1} = x^k + z$$

$$\text{where} \quad Mz = b - Ax^k.$$

In the conjugate gradient method we must maintain symmetry so we use

$$CAC^T y = Cb \quad \text{where} \quad x = C^T y$$

and we require CAC^T to be wellconditioned and C to have a nice sparsity pattern.

Whilst the method could be programmed in terms of y it is more usual to substitute into x space then the initial step

$$d_1 = \Delta x_1 = C^T C(Ax - b)$$

$$r_1 = Ax_1 - b; \quad z_1 = C(Ax_1 - b)$$

and at subsequent iterations

$$x^+ = x + \alpha d$$

$$r^+ = r + \alpha Ad$$

$$z^+ = z + \alpha CAd$$

$$d^+ = Cz^+ + \beta d$$

where $\alpha = - r^T d / d^T A d$ and $\beta = z^{+T} z^+ / z^T z$.

The usual choices of conditioning matrix on sequential machines are the Neumann or incomplete Choleski.

The Neumann conditions used the principle if $G = (I - A)$ then $(I - G)^{-1} = I + G + G^2 + G^3 \ldots$ and truncate this series for C.

On sequential machines these are not as fast as the incomplete Choleski factorisation, where a Choleski factorisation is done without fill in to prserve sparsity. On pipelines and parallel computers the presence of these matrices puts up the CPU time, as the algorithms are no longer readily adapted for parallel processing.

A typical set of results is that given by Blumenfield (1983)

Preconditioner	No. of Iterations	Pipeline Time
I	120	584
$I + G$	65	445
$I + G + G^2$	70	606
$I + G + G^2 + G^3$	45	493
Incomplete Choleski	37	1293

The problem of preconditioners for parallel and pipeline systems needs further research.

9. Conclusions

In this lecture I have tried to illustrate some of the considerations that must be taken into account when selecting an algorithm for use on a parallel processor architecture. I hope to have shown convincingly that the assumption that any sequential algorithm can be selected without careful consideration is false.

Of the Gaussian type methods the Gauss-Jordan version is recommended. for QR decomposition the Greedy parallel Givens method is the obvious choice. For iteration methods the Block Gauss Seidel is effective and

simple to program as is the conjugate gradient method. Great care needs to be taken when introducing preconditioning matrices into parallel computation.

References

1. Singh, P, An investigation into the prediction of the flow of viscous incompressible fluids, PhD thesis, Hatfield Polytechnic, 1983.

2. Ducksbury, P G, An investigation of the relative merits of optimisation algorithms on the ICL-DAP, PhD thesis, Hatfield Polytechnic, 1985.

3. Toint, P, An introduction to Quasi Newton Methods for large scale unconstrained nonlinear programming: University of Namur, Belgium, 1986.

4. Patel, K D, Parallel Computation and Numerical Optimisations, Annals of Operations Research 1, 135-149, 1984.

5. Rall, L B, Automatic Differentiation: Techniques and Applications, Lecture Notes in Computer Science No. 120, Springer Verlag, 1981.

6. Mohseninia, M, The use of the extended operator set of ADA with automatic differentiation and the truncated Newton Method, The Hatfield Polytechnic, NOC TR176, 1987.

7. Dixon, L C W, Dolan, P and Price, R C, Finite Element Optimisation: The use of structured automatic differentiation, The Hatfield Polytechnic, NOC TR175, 1986.

8. Byrd, R H, Schnabel, R B and Shultz, G A, "Using parallel function evaluations to improve Hessian approximation for unconstrained optimization," Technical Report CS-CU-361-87, University of Colorado, Boulder, 1987.

9. Sameh, A H and Kuck, D J, Parallel Direct Linear Equation Solvers - a survey, In M Feilmer (ed.) Parallel Mathematics, 1977.

10. Lord, R E, Kowalik, J S and Kumar, S P, Solving Linear Algebraic Equations on an MIMD Computer, J. of ACM, Jan 1983.

11. Kuck, D J, A survey of parallel machine organization and programming, Computing surveys 9, 29-59, 1977.

12. Sameh, A H and Kuck, D J, On Stable Linear System Solvers, J.A.C.M. 25, 31-91, 1978.

13. Sorenson, D, Lecture presented at XIV Mathematical Programming Symposium, Boston, 1985.

14. Cosnard,M and Robert, Y, Complexity of the parallel QR Decomposition of a Rectangular Matrix In Feilmeir, Joubart and Schendel (eds.) Parallel Computing 1985, Elsevier Science.

15. Modi, J J and Clarke, M R B, An alternative Givens ordering, Numer. Math. 43, 83-90, 1984.

16. Kowalik, J S, Kumar S P and Kangria, E R, An implementation of the fast Givens transformation on an MIMD computer, Applications Mathematicae, Polish Academy of Sciences, 1983.

17. Baudet, G M, Asynchronous Iterative methods for multiprocessors, JACM 2, 226-244, 1978.

18. Blumenfield, M, Preconditioning Conjugate Gradient Methods on Vector Computers In Feilmeier, Joubert and Schendel (eds.) Parallel Computer 83, North Holland, 1983.

END

DATE

FILMED

DTIC

JULY 88