Lecture Notes in Mathematics

Edited by A. Dold and B. Eckmann

1397

P. R. Turner (Ed.)

Numerical Analysis and Parallel Processing

Lancaster 1987

Springer-Verlag

PREFACE

THE SERC NUMERICAL ANALYSIS SUMMER SCHOOL

University of Lancaster

12 – 31 July, 1987

The essential aims of this third Numerical Analysis Summer School were much the same as those of the first two of these meetings in 1981 and 1984. The proceedings of the earlier events are published as LNM 965 and 1129. Each week of the meeting was largely self-contained although there was an underlying theme of the effect of parallel processing on numerical analysis running through the whole three week period. During each week there was opportunity for intensive study which could broaden participants' research interests or deepen their understanding of topics of which they already had some knowledge. There was also the opportunity for continuing individual research in the stimulating environment created by the presence of several experts of international renown.

This volume contains lecture notes for most of the major courses of lectures presented at the meeting. During each of the first and third weeks there was a ten lecture course and a related shorter course while the second week contained three related courses of various lengths.

Presented here then is an account of a highly successful meeting in what is becoming a regular series of SERC Numerical Analysis Summer Schools.

Acknowledgements

The most important acknowledgement is to the Science and Engineering Research Council who once again provided very generous sponsorship for the meeting. Their contribution covered all the organisational and running costs of the meeting as well as the full expenses of the speakers and the accommodation and subsistence expenses of up to twenty participants each week. A contribution towards the expenses of the meeting was also received from the British Gas Corporation.

I also wish to thank the SERC Mathematics Secretariate for their help in the planning of the meeting and especially Professor John Whiteman who acted as their assessor and as my advisor throughout. I should also like to acknowledge the help and encouragement of many of my colleagues in the University of Lancaster and especially Mrs Marion Garner who handled nearly all the secretarial work involved both in the planning and running of the meeting. As usual her help was invaluable.

Peter R Turner

Department of Mathematics,
University of Lancaster,
Lancaster LA1 4YL
England.

Mathematics Department,
United States Naval Academy,
Annapolis,
MD 21402, USA.

CONTENTS

PARALLEL COMPUTATION AND OPTIMISATION
A series of five lectures (July 1987)

Laurence C. W. Dixon
Numerical Optimisation Centre
The Hatfield Polytechnic

1. Introduction

The material presented in this paper was prepared as a series of five lectures to be presented at the SERC Summer School in Numerical Analysis held at the University of Lancaster in July 1987.

The intention in the series of lectures was to indicate the variety of parallel processing architectures then available and to discuss how the design of software for solving optimisation problems is affected by the various architectures.

The first lecture therefore reviewed briefly the architectures then available, particular attention being paid to those on which implementations of optimisation algorithms had been attempted.

The second and third lectures discussed the design of optimisation codes on parallel processing systems. One result of this discussion is the conclusion that if optimisation codes are to be efficiently implemented on parallel systems, then sets of linear equations must be efficiently implemented as part of the code.

The fourth lecture therefore discussed the solution of sets of linear equations on parallel processing systems. This mainly concentrated on the solution of dense sets of equations as the concurrent lectures by Alan George on the solution of sparse systems formed the main topic of that week.

The fifth lecture described the results we have achieved solving structured optimisation problems on the DAP. The lecture series concluded with a brief description of some of our experiences solving sparse sets of equations. This last topic is however omitted from this paper.

In presenting the series of lectures in this paper I wish to acknowledge the help and assistance of many colleagues at the Numerical Optimisation Centre, of our research students and of their sponsors, without their support the research described could not have taken place.

2. Parallel Computers and Parallel Computing

The last few years have seen the introduction of many different designs of parallel computers. This process has been encouraged by three similar but distinct considerations. The first need that has led to the development of parallel computers was the desire to be able to solve problems that require more data storage and/or take too long to solve on a single mainframe conventional sequential computer. The type of problem that still exceeds the capabilities of such computers is the time varying three dimensional solution of flow problems. To solve such problems we must either develop a faster sequential machine or develop a parallel machine. The choice is usually determined by the relative cost and until recently sequential machines have become faster at a quicker rate than parallel machines have been

developed so that at any time the largest fastest machines were sequential rather than parallel. This situation has changed for the first time recently and since (1985) parallel machines have been faster.

The second need that has led to the development of a very different type of parallel computer is the requirement to be able to solve more and more sophisticated on-line problems in real time. Here the availability of cheap microprocessors has led designers to wish to use more sophisticated algorithms in the control of systems thus improving their performance. The need to use more sophisticated algorithms has led to the need for including more computational power in on-line systems. This need can again be met by the use of a more powerful sequential chip or alternatively by the use of a number of less powerful chips in parallel. The rate of increase in power of cheap chips has to date been so great that the easier option of using a more powerful sequential chip has usually been chosen. It is, however, true that many on-line problems involve the continual computation of a number of separate but interactive tasks, that are artificially inter- weaved on a sequential chip by time slot sharing. Such problems, containing truly parallel tasks, are the third driving force behind the need for parallel computing.

These three problems are very different and obviously need different parallel hardware, however, the principles that are used to categorise the different classes of parallel computer are not based on size and cost but rather on the different levels and types of parallelism. Strictly each type of parallelism could be used in the design of hardware of all three sizes of computers though these have not all been implemented. In the next section the basic principles of each way of introducing parallelism will be described.

2.2 Parallel Computers

2.2.1 General Comments

One of the earliest classifications of parallel computers was given by Flynn (1972), who divided parallel systems in Single Instruction Multiple Data (SIMD) machines and Multiple Instruction Multiple Data (MIMD) machines.

SIMD machines use the principle that it is relatively easier to instruct a large number of units to do the same thing at the same time than it is to instruct the same number of units to carry out different individual tasks at the same time. Whilst it is easy to instruct thousands of units to simultaneously do the same task either in the army or the factory it is normally necessary in either to have a tree structure of command where one manager only instructs between five to fifteen subordinates himself if they are expected to act individually. Similarly, SIMD machines with over 1000 processors already exist but more versatile MIMD machines usually contain less than 10 processors.

As well as distinguishing between these two categories it is also necessary to distinguish carefully the level of tasks that are undertaken in parallel and the granularity (computational length) of those tasks that are being undertaken in parallel.

Other important considerations are the need to transfer data to and from the parallel processors. This can lead to conflict in accessing memory that can degrade performance. A similar problem is the frequent need to synchronise the arithmetic being performed on the processors to ensure that if processor A needs to use a new value of data being calculated by processor B then processor A does not do that calculation before processor B completes its task. This is known as the synchronisation problem and it can lead to very inefficient performance if it is not carefully considered in the design of codes. The synchronisation problem can usually be avoided on SIMD machines but is so important on MIMD machines that the possibility of using asynchronous algorithms has seriously been proposed by Baudet (1978).

2.2.2 Pipeline Machines

One of the most successful ways of speeding up arithmetic by introducing parallelism is known as the pipeline. Pipelines are designed to speed up the calculation of vector loops typified by

$c_i = a_i + b_i \qquad i = 1 \ldots n$

or $c_i = a_i * b_i \qquad i = 1 \ldots n.$

Let us consider the summation of two numbers

a = 1, 123, 765

b = 2, 410, 123.

We start by adding the units, then the tens, then the hundreds, etc. so when performing in decimal arithmetic on this example 7 virtually independent additions are undertaken that are only linked by the need to be able to carry forward the overflow from one operation to the next. This implies that one cannot be started until the previous one is complete. So on a decimal machine if the time for one such operation was t, then the time for performing $a_i + b_i$ would be 7t and on a sequential machine the time for the loop would approximate 7nt. Now suppose we were to put P = 7 such processors in a line and let the first processor handle the units, the second processor the tens, the third the hundreds, etc. Then the first processor can start on the second sum while the second processor is still working on the first so that all the loop will be completed in (n + 6)t units. The speed up is therefore (n+6)/7n, notice that as n gets large this is roughly 1/7 i.e. proportional to 1/P but that there is a start up time that can make the gain negligible for small values of n.

Obviously the operation of any pipeline machine is more sophisticated than this, to increase efficiency pipelines can be chained allowing the arithmetic in a second loop to be started before the arithmetic in the first is complete. Another complication but this time one that normally has an adverse effect is that due to memory access and data structure considerations there is often an upper limit N on the length of loop n that can be pipelined.

If we let I(n/N) be the first integer greater than n/N where n is now the number of data in the desired loop then this degrades the time for our example to approximately I(n/N)(N+6)t units.

The three most commonly used parallel computers the CRAY 1, CYBER 205 and the Floating Point Systems APB series all utilise the pipeline principle.

In these machines the level of the parallelism is very low being within the basic arithmetic operations. This means that sequential high level codes could in principle be directly transferred to pipeline machines, and indeed this is frequently done, but for full benefit to be obtained sequential codes frequently have to be re-organised to introduce more N loops that can be pipelined and chained.

This principle is undoubtedly the most widely used parallel method for solving large problems. While it could, in principle, also be used for real time computing, no machine based on it smaller than the Floating Point system series is known to the author.

2.2.3 Arrays of Processors

The second most common type of SIMD system consists of arrays of processors; typically a large number of identical processors are arranged in a 2 dimensional grid with processors placed at the nodes of equally spaced orthogonal lines. The processors are then connected by fast data links along these orthogonal lines and also usually by fast links connecting the ends of these lines. If we introduce axes parallel to these lines it is natural to refer to a processor at position $x_1 = i$, $x_2 = j$ as processor $P_{i,j}$ then if we wish to do typical two dimensional matrix algebra

$C(i,j) = A(i,j) + B(i,j)$

for all values of i,j, then all these operations can be performed in parallel by assigning the calculation of C(i,,j) to P(i,j). The level of parallelism is therefore higher than in the pipeline as it is arithmetic operations that are being performed in parallel, rather than each operation being split into parallel subtasks. For efficiency it is desirable to have a long sequence of such matrix operations before having to transfer data to the host to undertake scalar arithmetic.

Typically if we are considering a square matrix i = 1 ... n j = 1 ... n where $n^2 > P$ the number of processors then the time for performing such a sequence varies as shown in Figure 1.

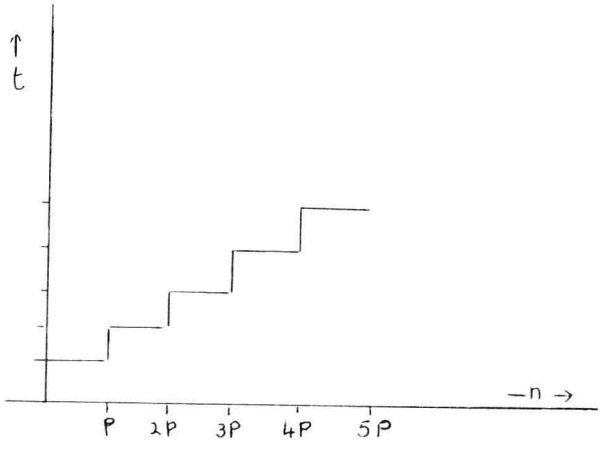

Figure 1

The two most well-known examples of systems based on arrays of processors are the ILLIAC IV and the ICL-DAP

2.2.4 Data Flow

When considering SIMD machines we considered one example of a small grain and one of an intermediate grain system. The data flow principle is again based on dividing arithmetic into parallel tasks at the basic arithmetic operation. A data flow machine will contain a number of processors and a free processor commences the next arithmetic operation for which all the data is available. Consider the statement

$y = 2x^3 + 3x^2 + 4x + 5.$

On a sequential machine this involves 5 multiplications and 3 additions i.e. 8 sequential steps, however, if we allow parallel operations and 3 processors we can construct an operation graph in which there are only 4 sequential steps. This would therefore only take half the time and of course if we had defined other tasks then some processors could already have started them.

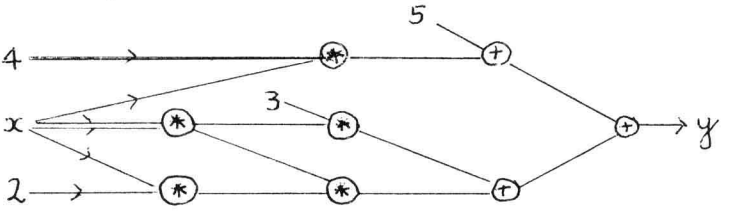

Figure 2

There is as yet no machine available commercially based on the dataflow principle, but considerable research into such machines is currently being undertaken. Such machines should be very powerful. I have seen such graphs describing the solution of a set of sparse 10 x 10 equations in under 20 sequential steps.

2.2.5 Parallel Networks

The most common way of achieving parallel MIMD computing is to connect separate processors in a network and allow them to co-operate on a task.

The pioneer work in this area took place at Carnegie-Mellon with the design of the CM* and Ccmp machines (7). This was followed by a large number of small systems containing 4 or 5 processors closely linked together. A typical example is the NEPTUNE system built by David Evans' team at Loughborough University, Barlow (2). This consisted of 4 Texas Instruments microprocessors. Experience showed that with such a system of 4 processors it was relatively easy to code least squares optimisation algorithms and global optimisation algorithms that ran more than 3 times faster than on a single such processor, Patel (1983). Simulation studies showed, however, that with this particular structure, data transfer and data access conflicts increased rapidly with P, the number of processors, and that there would be virtually no benefit in combining more than 10 processors in that mode, McKeown (1980).

In 1985 the most powerful MIMD machine in use was the CRAY XMP which links 2 CRAY 1's together. Cray have recently announced the more powerful CRAY 2 which will

contain at least 4 CRAY 1's linked. These numbers again emphasise the small value of P envisaged in most MIMD systems.

Other geometries are being investigated, for instance, at Madison University a ring of Vax's is being built which should be a very powerful combination, whilst IBM and Floating Point Systems have just (1985) announced the setting up of 4 Parallel Computing Research Centres equipped with an IBM mainframe with 10 Floating Point Systems pipeline machines connected to it. This system therefore combines the MIMD mode with the pipeline principle at the lower level. It should be a very powerful research instrument.

The pace at which parallel computers are being developed can be judged by contrasting the above description which was reasonably accurate in 1985 with the situation at the UNICOM seminar [3] held in London in December 1986.

At that Seminar the producers of small parallel computers were allowed to display their current machines and twelve elected so to do. These machines ranged in size and cost from relatively cheap small transputer systems through more sophisticated MEIKO transputer systems which are simple MIMD systems with between 4 and 40 transputers; to the new ADM version of the DAP (1032 SIMD faster processors) and the rival Intel hypercube with its different network structure. Also on display were the Sequent parallel computer that contains between 2 and 30 National Semi conductor 32032 microprocessors and is capable of running both ADA concurrent tasking packages and by simple code modification of running loops in Fortran in parallel. The Alliant machine containing more powerful processors was also on display. It is very difficult to keep up with hardware developments in this area and the software that is available to run on them. In consequence any brief description of this type now soon is out of date.

In this lecture series I will not be considering pipeline machines or algorithms; but will discuss our experience on MIMD and SIMD systems. In 1984 Dew contrasted the different implications of pipeline and other parallel systems in the following words "The need to perform ever more complex scientific computations is clearly illustrated by the success of the new generation of vector processors like CRAY. The speed-up achieved by these has in the main been brought about by architectural improvements (e.g. pipelines) and the development of compilers to map sequential programs onto vector architecture. Although new algorithms are being developed to exploit pipelines, in general their introduction has had little effect on underlying numerical algorithms. This is not the case for arrays of processors like the DAP, where new approaches are required. Then, the problem, architecture and algorithm must be more closely related."

It is this challenge to design parallel software that mimics the parallelism in both the problem and the hardware that is the exciting aspect of parallel processing.

References to Section 2

1. Barlow, R H et al, A guide to using the Neptune processing system, Loughborough University of Technology.

2. Baudet, G, The design and analysis of algorithms for asynchronous multi-processors, PhD Dissertation, Carnegie Mellon University, 1978.

3. Proceedings of Major Advances in Parallel Processing, Unicom Seminar, 9-11 December 1986.

4. Flynn, M, IEEE Transactions on Computers, Vol C21, No 9 Sept 1972, pp 948-960.

5. McKeown, J J, (1980), Simulation of a parallel global optimisation algorithm. NOC TR 109, March 1980.

6. Patel, K D, (1983), Implementation of a parallel (SIMD) modified Newton Algorithm on the ICL DAP, NOC TR 131, 1983.

7. Wulf, W A & Bill, C G, C.MAP - a multi mini processor, AFIPS proc 1972, FJCC Vol 41, AFIPS press, pp 765-727.

3. Solving Optimisation Problems on Parallel Processing Systems

In this lecture 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, while the second class will be discussed mainly in the final lecture. The other two will not be discussed in any depth in this series.

3.2 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 three distinct ways.

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 which is undesirable.

Approach B

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

Let us first consider two examples of Approach A.

Approach A Example 1 A H O Brown (1976) [1]

Suppose we wish to optimise the design of an aero engine so that the fuel used to cross the Atlantic is a minimum. Then each design $x^{(k)}$ implies a calculation of the performance of the engine in a number of states, e.g. take off, climb, subsonic cruise, transonic flight, supersonic flight, de-acceleration, descent, hold over final airport, final descent and landing, a total of eleven effectively separate tasks. These could be performed in parallel but unless considerable care is taken to divide the computation into roughly P equal tasks many of the processors will be idle for most of the time and hence the processor use will be inefficient. Such a calculation also requires an MIMD machine as each subprocess is different.

Approach A Example 2

A simpler example would be the least squares minimisation problem

$$\text{Min } F = \sum_{i=1}^{mP} s_i^2(x)$$

where $s_i(x)$ is the difference between an experimental data value y_i and the model

value at say t_i

i.e. $s_i(x) = y_i - m(x, t_i)$.

Now if we divide the mP data points in P equal sets each parallel processor does the same SIMD task. This has been implemented on both the MIMD Neptune system [McKeown (2)] and on the ICL-DAP with the expected speed up. [Sargon, Chong & Smith (3)].

Before considering approach B let us now consider the expected speed up.

3.3 Performance Evaluation

Measuring the performance of a parallel system must be dependent on the type of parallel system used.

The Neptune system at Loughborough consisted of P = 4 small microprocessors. The only logical comparison is with itself i.e. use of P processors or 1 processor.

In contrast the DAP consisted of 4096 processors and purported to compete with a mainframe. We therefore compared it with the DEC 1091 which was The Hatfield Polytechnic mainframe at that time.

3.3.1 MIMD systems

Two concepts are usually used for measuring the performance of a parallel system, they are "speed up" and efficiency.

Let $\tau(p)$ be the processing time using P identical processors. Then the "speed up S" is defined as

$$S = \frac{\tau(1)}{\tau(P)}$$

and the efficiency

$$E = \frac{S}{P} \leq 1$$

Ideally we might expect the speed up ratio S to be P and hence the efficiency E = 1. In general, however, some degradation must be expected. The main factors that contribute to this degradation are:

a) at the system level

(1) the actual processing speeds of the processors differ

(2) input/output interrupts

(3) memory contention

(4) data transfer time between processors

b) at algorithmic level

(1) synchronisation losses, if P tasks are to be performed all usually wait for the slowest

(2) critical section losses, if the code requires all processors to access, say, global memory at once.

3.3.2 SIMD system

For the ICL-DAP we measured the speed up by

$$S = \frac{\text{processing time on Dec 1091}}{\text{processing time on ICL-DAP}}$$

for a number of standard test functions Patel (1983) reported S = 20. This was on a simple problem where 4096 function evaluations were performed on both machines. This emphasises the slow nature of the individual processors on the DAP which individually must be approximately 200 times slower than the Dec 1091. Later tests gave a value of S = 60 indicating that such comparisons cannot be expected to remain constant with time.

These ratios must be borne in mind in considering the later comparisons.

3.4 Optimisation Algorithms Approach B

For unconstrained optimisation problems it was generally accepted c.1983 that there were 4 broad categories of algorithms

(1) Modified Newton Methods $2 \leq n \leq 5$

(2) Variable Metric Methods $5 \leq n \leq 120$

(3) Conjugate Gradient Methods $n \geq 60$

(4) Conjugate Direction Methods $n \leq 30$

where the choice of algorithm was mainly determined by the dimension of the problem n.

The question that naturally arises is that if we have P processors how would the choice be effected.

3.4.1 Newton Raphson code

The modified Newton Method we chose to investigate was due to Mifflin [5].

Mifflin's Method

In Mifflin's Method at iteration k we have

Step 1 Calculate $F(x^{(k)})$ at $\frac{1}{2}n(n-1)$ places to estimate $g = \nabla F$ and $G = \nabla^2 F$ by differences. If $P \geq \frac{1}{2}n(n-1)$ these can be done in parallel.

Step 2 Estimate

$$g_i = (F(x + h_{ai}) - F(x - h_{ai}))/2h$$

$$G_{ij} = (F(x + h_{ai} + h_{aj}) - F(x + h_{ai}) + F(x + h_{aj}) - F(x))/h^2.$$

Again these are parallel computations and if $n \leq 64$ the matrix G_{ij} can be stored in a single field on the DAP.

Step 3 If $\max |g_i| \leq \varepsilon$ stop.

Step 4 Solve $Gd = -g$.

[On the DAP; DAP library routine FØ4GJNLE64 was used].

Step 5 Find α so $F(x^{k+1}) < F(x^k)$

where $x^{k+1} = x^k + \alpha d$.

The line search is essentially sequential. To evaluate 4096 points along the line would be unhelpful, so a 4D search was introduced

$$x^{k+1} = x^k + \sum_{i=1}^{4} \alpha_i p_i$$

Step 6 Return to step 1.

This code was implemented on the ICL-DAP by Patel [4] who reported the following comparison.

	N-R	V-M	C-G	DAP
Quadratic	15.00	1.8	1.2	1.2
	16.60	2.1	1.1	1.2
Extended Rosenbrock	145	103	8.4	4.9
	141	80	11.4	28.2
Extended Powell	112	53	5.9	2.5
	134	41	11.1	1.9
Trigonometric	F	66	37.6	13.9
	2604	286	78.7	8.6
Extended Box	4181	1161	137.8	107.9
	7195	3194	354.0	323.9

A comment on these results seems necessary. The problems were all in 64 dimensions and were all extended versions of standard problems. In 64 dimensions we would expect V-M and C-G codes to outperform N-R sequentially and they do. However on these extended standard problems the Hessian only has 4 distinct eigenvalues (except for the Trigonometric problem) and this further favours the C-G code. On speed up time the DAP implementation is usually more than 20 times faster than N-R but is no real improvement over the C-G code that would normally be used in this dimension.

Before leaving this algorithm I would like to comment that it is necessary to use a parallel equation solver at step 4 otherwise if 95% of the time is spent in evaluating function evaluations and 5% on overheads (mainly the equation solver) then

$$S < \frac{100}{5+95/P} < 20 \text{ all } P!$$

3.4.2 Variable Metric code

On the DAP system the variable metric code which only evaluate ∇F could do so using central differences in one step if

$2n + 1 \leq P$.

However the need to store an n x n Hessian matrix would seem to make such an approach undesirable. Indeed to store H in one matrix field limits $n < 64$ and then the full Newton algorithm can be implemented as we have just seen.

For other architectures i.e. for small values of P and larger store availability the possibility of updating the matrix H for P parallel steps d_j has been examined by Straeter [6] who gave the following scheme

$$r_j = H^{k-1} y_j - d_j - \sum_{i=1}^{j-1} \frac{r_i^t y_j}{(r_i^t y_i)} r_i \qquad j = 1 \ldots P$$

$$H^k = H^{k-1} - \sum_{j=1}^{P} \frac{r_j r_j^T}{y^{jT} r_j}$$

Here $y_j = \nabla F(x + d_j) - \nabla F(x)$.

The precise way of modifying the line search would depend upon the value of P.

3.4.3 The conjugate gradient code

On the DAP system the conjugate gradient code which also only requires ∇F can be implemented using central differences in one step if

$2n + 1 \leq P$.

Again as it only needs to store a few n vectors this can be implemented on the DAP easily for values of $n \simeq 4096$. It is therefore ideally suited to solve problems rather larger than those we usually consider.

This observation led to the study on finite element optimisation that will be the subject of section 6. The results will be discussed there.

3.4.4 Summary

Straeters variable metric method is intended for the range $P < n$; the conjugate gradient method is ideal if $P \sim n+1$ and the Newton Method if $P \sim (n^2 + 5n + 2)/2$. In a recent paper Byrd, Schnabel and Shultz (1987) discuss a modified variable metric method for the range $n+1 < P < (n^2 + 5n + 2)/2$. Codes have therefore been suggested for a wide range of p/n.

3.5. The Truncated Newton Algorithm

At this point in our research the direction we were following was influenced by the announcement of the Truncated Newton Algorithm by Dembo and Steihaug [7].

This was quickly implemented at Hatfield by Price [8]. It consists of essentially the following steps. It is still iteractive in that $x^{(k+1)} = x^{(k)} + \alpha d$ but it differs dramatically in the way d is calculated.

In the Newton Method d is obtained by solving the set of equations

$Gd = -g$.

In the truncated Newton method this set of equations is approximately solved by a few iterations of the conjugate direction method for linear equations.

In particular we may note that if u_i is the prediction of d after i steps of the conjugate direction method then

(1) $||u_i|| > ||u_{i-1}||$

(2) If $Q(u) = \frac{1}{2}u^T G u + g^T u$ then $Q(u_{i+1}) < Q(u_i)$.

So each prediction is larger than the previous one and reduces the quadratic approximation Q of F.

In the truncated Newton method there is a trust region of diameter D and the conjugate direction method is stopped if either

(1) $||u_{i+1}|| > D$

and indeed the step in the conjugate direction p_i is reduced to $\alpha_i p_i$ where

$||u_i + \alpha_i p_i|| = D$,

or (2) $$\frac{r_{i+1}^T r_{i+1}}{g_k^T g_k} < \text{Min}\left(\frac{0.1}{k^2}, g_k^T g_k\right).$$

The first prevents the step getting too large whilst the second prevents too accurate a solution of the set of equations far from the optimum.

In the Hatfield implementation we included a conical trust region to ensure Wolfe's condition [9] for effective descent is also satisfied namely

(3) $(u_{i+1}{}^T g_k) < -0.1\ ||u_{i+1}||\ ||g_k||$.

Again if necessary the step $u_{i-1} + \alpha p_i$ is chosen so that this is an equality.

The code is therefore

Step (1) Select $x^{(0)}$, ε, k_{max}, P_o: $k = 1$

(2) Calculate $F(x^{(k)})$, $g = \nabla F(x^{(k)})$

(3) Stop if $g^T g < \varepsilon$ $k > k_{max}$

(4) Use the conjugate direction code

(4.1) Set $u_i = 0$, $r_1 = p_1 = -g$, $i = 1$

(4.2) Calculate $\alpha_i = r_i{}^T r_i / p_i{}^T G p_i$

(4.3) $u_{i+1} = u_i + \alpha_i p_i$

(4.4) $r_{i+1} = r_i + \alpha_i G p_i$

(4.5) Stop if any of the 3 truncating conditions is satisfied and go to (5)

(4.6) $\beta = r_{i+1}{}^T r_{i+1} / r_i{}^T r_i$

(4.7) $p_{i+1} = + r_{i+1} - \beta p_i$

(4.8) Goto (4.2) next i

(5) Evaluate $F(x + u)$

(6) Fit a parabola to $F(x)$, $g^T u$, $F(x + u)$ if the predicted step is outside $0.8 < \alpha p < 1.2$ evaluate $F(x + \alpha p u)$

(7) Test better of $F(x + u)$, $F(x + \alpha p u)$ against Wolfe's tests II and III, Dixon [9] and if necessary do an Armijo line search [10].

Let α^* be an accepted value of α.

(8) Put $x^{k+1} = x^k + \alpha^* u$

if $\alpha^* \geq 1$ put $D_{k+1} = 2\ D_k$

if $\alpha^* < 1$ put $D_{k+1} = 1/3\ D_k$

return to (2) next k.

In this code the term Gp_i occurs at steps 4.2 and 4.4. Dembo proposed calculating this approximately by differences

$Gp_i = \{g(x + \sigma p_i) - g(x)\}/\sigma$.

This saves the storage of G at the expense of gradient calls. The additional gradients do not have to be stored.

The tests shown in Table 1 demonstrate that this method outperforms the N-R(E04KDF), V-M(OPVM) and C-G(OPCG) codes on all the test problems with $n > 4$.