EXPERIENCE RUNNING OPTIMISATION ALGORITHMS ON PARALLEL PROCESSING SYSTEMS

L C W Dixon, K D Patel and P G Ducksbury
The Numerical Optimisation Centre
The Hatfield Polytechnic
P.O. Box 109
Hatfield, Herts. AL10 9AB
United Kingdom

ABSTRACT

The availability of parallel processing machines makes it possible to envisage solving four types of numerical optimisation problems that still present difficulties on fast sequential machines. These four types of problems are described, then two parallel processing machines, ICL DAP and NEPTUNE are briefly described, and ways of mapping the problems onto ICL DAP and NEPTUNE are discussed. Finally, our experience in implementing parallel algorithms on ICL DAP and NEPTUNE are detailed.

1. INTRODUCTION

We are concerned with solving the optimisation problem

$$\min f(\underline{x}) , \qquad \underline{x} \in R^n$$

subject to simple upper and lower bounds, i.e. $l_i \leq x_i \leq u_i$.

There exists a number of sequential algorithms with superlinear convergence, based on the iterative scheme,

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

where $\underline{p}^{(k)}$ is the search direction and α the step size. These algorithms can solve most optimisation problems. Given this situation the question arises as to why then should we be interested in introducing the parallel processing concept in numerical optimisation. The main reasons are:

a) The processing time for solving some of the industrial optimisation problems is too long,

b) There are some problems that cannot be tackled because their size leads to storage difficulties on sequential systems,

c) People only tend to pose optimisation problems that they feel might be soluble.

By introducing the parallel processing concept to numerical optimisation we hope to reduce some of the difficulties encountered in the sequential approach.

The four different situations where we feel improvements are most likely in the solution of optimisation problems with the currently available parallel processing machines are:

a) Small dimensional problems

Small dimensional problems, n < 100, where the cost of computing the objective function/gradient vector/Hessian matrix greatly exceeds the overheads in the optimisation part of the algorithm.

b) Large dimensional problems

Large dimensional problems, n > 2000, where the combined processing time and storage requirements causes difficulties.

c) On-line optimisation

On-line optimisation problems, like the optimisation of car fuel consumption, cannot always be solved using the existing sequential algorithm because of the time factor and in these problems a saving of the order of four or five could be sufficient.

d) Multiextremal global optimisation problems

Problems where the objective function has many local minima. For these problems the sequential algorithms for locating the global minimum are still relatively unsatisfactory and expensive in computer time.

Later in this paper we will consider each classification in more detail, describe parallel algorithms and present numerical results. Before we consider the interaction of the four classifications mentioned above and the parallel processing computers, we will briefly described the parallel processing computers available to us.

2. CLASSIFICATION OF PARALLEL COMPUTERS

We classify parallel computers as either SIMD (Single Instruction Multiple Data) or MIMD (Multiple Instruction Multiple Data).

2.1 SIMD systems

We subdivide the SIMD machines as vector and array computers. The difference is based primarily on the way data is communicated to elements of the system.

2.1.1 Vector processors

Vector processing is the application of arithmetic and logical operators simultaneously to components of vectors. This leads to a straightforward reduction in time at the arithmetic and logical operations stage whenever a vector processing system is used. Examples of vector processors are CRAY-1/S and CYBER 205 systems. Both contain pipelined vector arithmetic processors capable of performing operations on arrays at very high speeds. In one strict sense, pipelined computers are not parallel computers as the parallelism occurs within the fundamental arithmetic operations. However, their efficient use depends heavily on the redesign of algorithms to make efficient use of the parallel arithmetic option.

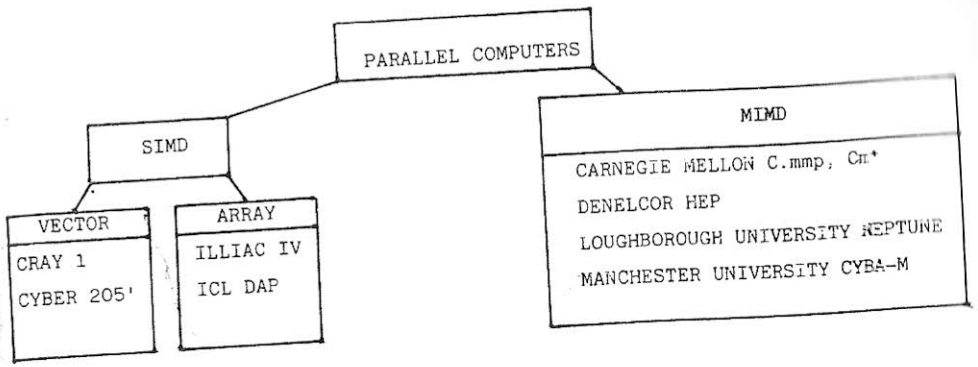


Figure 1. Classification of parallel computers

Most of the research in optimisation using vector processors is in the area of solving very large optimisation problems. The underlying optimisation approach is the Conjugate Gradient algorithm. The principle is to 'vectorise' the arithmetic within conjugate gradient algorithms. To make an efficient use of a vector processor, an algorithm must be arranged to do nearly all its arithmetic and logical operators on long vectors. Various researchers, Rodrigue and Greenbaum [1], Rodrigue et al [2], Schreiber [3], have reported their experience in implementing conjugate gradient algorithms on 'vector' machines.

2.1.2 Array processors

An SIMD array machine is defined as a computer with a single master control unit and multiple directly connected processing elements (see Figure 2). Each processing element is independent, i.e. has its own registers and storage, but only operates on command from the master control unit. Data access is from its own local memory and that of its nearest neighbour. Each processor carries out the same instruction as all the others but on its own specific data set. Because of the simplicity of each processing element, the number of processors that can be connected together in an SIMD array sense can be very large.

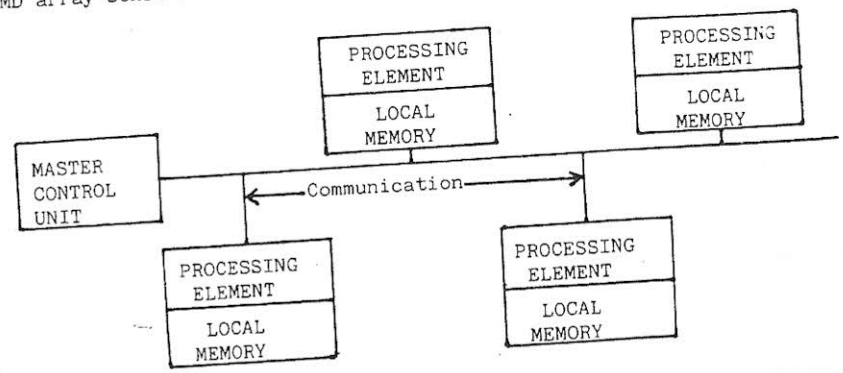


Figure 2. Array processor

Some of the application areas for an array processor are:

a) large problems such as weather data processing,

b) problems that have a large number of independent data sets.

The earliest SIMD array machine was the 64 processor ILLIAC IV. The most recent one in the UK is the 4096 processor - ICL Distributed Array Processor (DAP).

As one of the parallel systems we have used for the implementation of parallel algorithms is the ICL DAP, we will briefly describe the ICL DAP.

The ICL DAP at Queen Mary College, London, comprises 4096 elements each originally with a local store of 4K. (This has now been increased to 16K). All the processors obey a single instruction stream broadcast by a Master Control Unit. Processors are arranged in a 64 x 64 matrix form and each processor has access to the four neighbouring elements and to their stores.

A high level, special purpose language, called DAP-FORTRAN, is available on the DAP; this is an extension of FORTRAN with additional features that enable parallel algorithms to be expressed naturally and efficiently. DAP-FORTRAN has about 50 built-in functions for manipulation of vector and matrices. The implication of the hardware structure is that 64 x 64 matrices, 4096 long element vectors and 64 element vectors can be processed as single entities.

For example the DAP-FORTRAN declaration

```
REAL A(,), B(,), G()
```

is equivalent to the standard FORTRAN declaration

```
REAL A(64,64), B(64,64), G(64)
```

and the DAP-FORTRAN statement:

```
A = A + B
```

is equivalent to the standard FORTRAN loop:

```
DO   10   I = 1,64
DO   10   J = 1,64
10 A(I,J) = A(I,J) + B(I,J).
```

The statement A = A + B represents one operation performed simultaneously in all 4096 processing elements. The example below illustrates how some of the DAP-FORTRAN built-in aggregate functions can be utilised in numerical optimisation.

Consider a 64-dimensional Rosenbrock function

$$f(\underline{x}) = \sum_{i=1}^{32} 100(x_{2i-1}^2 - x_{2i})^2 + (1 - x_{2i-1})^2$$

The first four components of the gradient vector are:

$$g_1(\underline{x}) = 400x_1(x_1^2 - x_2) - 2(1-x_1)$$

$$g_2(\underline{x}) = -200(x_1^2 - x_2)$$

$$g_3(\underline{x}) = 400x_3(x_3^2 - x_4) - 2(1 - x_3)$$

$$g_4(\underline{x}) = -200(x_3^2 - x_4).$$

The DAP-FORTRAN code for the analytic gradient vector is

```
G = MERGE(400.0*X*(X**2 - SHLP(X,1)) - 2.0*(1.0 - X), - 200.0*(SHRP(X**2,1) - X),
    .NOT.ALT(1)
```

where G and X are declared as

```
REAL G(), X().
```

The components of G, the gradient vector, are computed in parallel.

We briefly summarise the effect of the DAP-FORTRAN built-in aggregate function used in the gradient evaluation.

SHLP - This function returns a vector value that is equal to the first argument shifted a number of places to the left, using plane geometry. The number of places by which the vector is to be shifted is given by the second argument.

SHRP - as for SHLP except that the vector is shifted to the right.

MERGE - returns a vector (or matrix) result, components of which are selected from corresponding components of the first or second argument, depending on whether the corresponding component of the third argument is .TRUE. or .FALSE. respectively.

ALT - function takes a single integer scalar argument and returns a logical vector value. If the value of the argument modulo 64 is i, the logical vector will have its first i components .TRUE, and so on in alternation until all the components of the vector have a value.

e.g. let $Z = (z_1, z_2, z_3, z_4)$

$Y = (y_1, y_2, y_3, y_4)$

SHLP(Z,1) = $(z_2, z_3, z_4, 0)$

SHRP(Y,1) = $(0, y_1, y_2, y_3)$

MERGE (Z,Y, ALT(1)) = (y_1, z_2, y_3, z_4).

Details of all the DAP-FORTRAN built-in aggregate functions can be found in the ICL DAP manuals [21].

2.2 MIMD systems

An MIMD system is defined as a series of processors (minis/micros) working independently in parallel. The number of processors cannot reach the same order as an Array processor because of the complexity in communications between processors and the

difficulties introduced by data access from a global data set. Early examples of MIMD systems were the C.mmp and Cm* designed at Carnegie-Mellon University, Pittsburgh. A more recent example is the Heterogeneous Element Processor (HEP) of Denelcor HEP which typically consists of from one to sixteen processors.

Another parallel system we have used for the implementation of parallel algorithms is the LUT NEPTUNE, which is briefly described below. The system contains 4 processors (Texas Instruments 990/110 minicomputers) and the current configuration is shown in Figure 3. Each processor has access to 96Kb of memory (local memory)

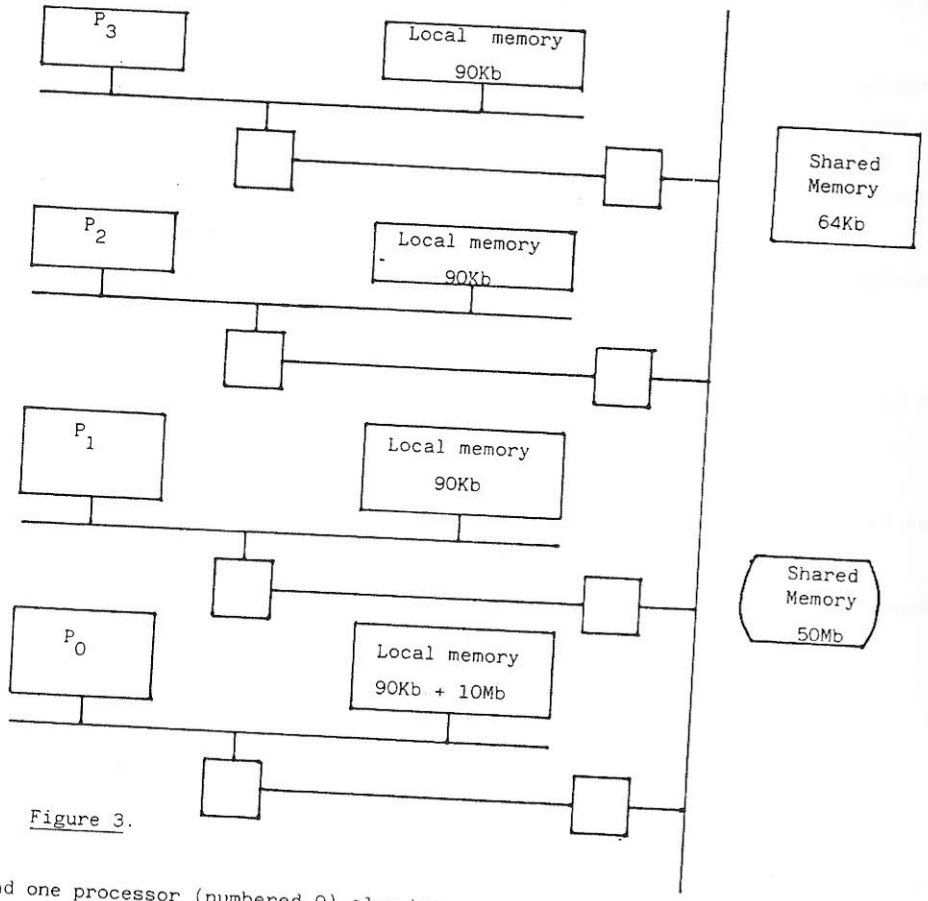


Figure 3.

and one processor (numbered 0) also has access to 10Mb of memory on a disc drive. There is also 64Kb of shared memory and 50Mbdisc drive. Hence each processor has a nett memory of 160Kb. The 50Mb drive can be accessed by each processor with disc interrupts being sent to them all.

The software used for executing a program on the NEPTUNE multiprocessor system is standard FORTRAN and some pseudo-FORTRAN syntax constructs.

The creation and termination of paths with identical code is defined using


```
$DOPAR   10 I = 1, N1, N2, N3
'code'.

10 $PAREND.
```

The creation of paths with different code is defined using

```
   $FORK 10, 20, 30, 40
10 'code 1'
   GOTO 40
20 'code 2'
   GOTO 40
30 'code'
40 $JOIN
```

Creation of paths with the same code and with each processor forced to execute the code once and once only is defined using

```
   $DOALL 10
   'code'.
10 $PAREND.
```

Shared data is defined using

```
   $SHARED variable list.
```

All other data, including program code, is held in local memory.

For critical sections of the program the user has available up to 8 'resources' which can only be owned by one of the processors at any one time. Resources used are declared using

```
   $REGION list of names
```

and claimed or released using

```
   $ENTER name or $EXIT name.
```

A fuller description of the NEPTUNE multiprocessor system can be found in Evans, et al [20].

New MIMD architectures, called data flow computers, which depart dramatically from the classical Von Neumann architecture are in the design stage. The data flow approach to computing involves evaluating statements in a program as and when their input operands become available. There has been a lot of theoretical work on the data flow architecture. Texas Instruments designed, built and benchmarked a four laboratory model of a data flow computer to assess the benefits that might arise, Sauber [5]. The main conclusions from their experience in using a data flow computer were:

a) given sufficient program parallelism, the processing power grows linearly with the number of processors,

a) given sufficient parallelism, all execution overhead is overlappable with useful work, and

b) a computer can find the necessary parallelism in ordinary programs, expose it, and then partition it automatically.

It is too early to say how numerical optimisation algorithms can benefit from data flow computers and no data flow system was available for our use.

The design of parallel algorithms for SIMD and MIMD requires two quite distinctive approaches. The main difference is, for a SIMD system we require that tasks should be identical, for a MIMD system we can dispense with that requirement. For a MIMD system we can also dispense with the requirement that tasks should be synchronous, at the cost of memory congestion and convergence problems.

Other distinctions will become apparent when we describe the parallel algorithms.

3. PERFORMANCE MEASUREMENT

Measuring the performance of a parallel algorithm is dependent on the type of parallel system used.

3.1 MIMD system

Two concepts available for measuring the performance of theoretical parallel algorithms are 'SPEED-UP' and 'EFFICIENCY'. They are defined as follows:

Let $\tau(P)$ be the processing time using P identical processors. Then the 'speed-up' factor over a single identical processors is

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

and the 'efficiency' is

$$E_p = \frac{S_p}{P} \leq 1$$

Ideally we might expect the 'speed-up' ratio, S_p, to be p and hence the efficiency E_p to be unity. In general, however, some degradation must be expected. The main factors that contribute to this degradation are:

a) at the system level:
   i) the actual processing speed of the processors differ,
   ii) input/output interrupts,
   iii) memory contention,
   iv) bottleneck in the data transfer.

b) at algorithmic level:
   i) synchronisation losses, if, say, p tasks are to be performed, all must wait for the slowest,
   ii) critical section losses, if many processors are trying to access the global data set at a particular instant, then only one processor can get such access at any instant, so while that processor is accessing the global data set most of the other processors are idle. This implies a time delay. We can minimise the waiting time by ensuring that access to the global data set is kept to a minimum.

3.2 SIMD system

The 'speed-up' and 'efficiency' ratios defined above are meaningless concepts for a SIMD type architecture. the performance of a parallel algorithm, for the SIMD type architecture is usually measured in relation to a sequential system. The equivalent sequential algorithm is on a sequential system. The performance ratio, τ , is given by:

$$\tau = \frac{\text{processing time on a sequential system}}{\text{processing time on a parallel (SIMD) system}}$$

This does, of course, depend on the sequential system chosen as the basis - in the results quoted in this paper the DEC system 1091 at The Hatfield Polytechnic was chosen. An additional complication in any such comparison arises because the optimum parallel algorithm for an SIMD system often differs considerably from the optimum sequential algorithm; and the choice of sequential code can, therefore, affect the result. In particular, should the best sequential code be used or one similar in principle to that implemented on the MIMD system.

4. INTERACTION OF PROBLEM CLASSIFICATION AND PARALLEL PROCESSING

Now let us consider the interaction of three of the four classifications, mentioned in Section 2, with the type of parallel processing computers we have considered. In particular, we will concentrate on the SIMD ICL DAP and MIMD NEPTUNE systems.

4.1 : Small expensive problems

Calculating function in parallel

Consider the situation of an extreme class of industrial problem in which, when attempting a solution by a sequential algorithm, the time spent in the function/constraint evaluation routine is, say, over 1000 times as long as that spent in the optimisation part of the algorithm.

A particular example of this is the case of optimisation of aircraft fuel engine performance subject to noise constraints and performance specifications. Details can be found in A H O Brown [6] of Rolls Royce Ltd. To parallelise such a problem we look for parallel features in the function evaluation routine. there is an obvious feature in the performance specification which applies to eleven different in-flight conditions and they can be computed in parallel. As each parallel task is independent, this approach is more applicable to an MIMD system than to an SIMD system.

When the function evaluation can be separated into a number of identical tasks as in a data fitting problem

$$\min \quad F(\underline{x}) = \sum_{k=1}^{M} S_k^2(\underline{x})$$

where each $S_k(x)$ is an identical calculation using different data, then we have an obvious calculation where (if $M \leq P$, the number of processors) an SIMD machine can be used.

Sargan, Chong and Smith [4] have solved large scale nonlinear econometric models on SIMD DAP by separating the objective function into identical tasks.

The efficient sequential approaches for solving small dimensional unconstrained optimisation problems are:

Modified Newton techniques: $2 \leq n \leq 5$
Variable Metric techniques: $5 \leq n \leq 120$
Conjugate Gradient techniques: $n \geq 120$
Conjugate Direction techniques: $n \leq 30$.

Nearly all efficient methods for solving unconstrained optimisation require the gradient vector $\underline{g}$ to be evaluated at $\underline{x}^{(k)}$. for some very expensive industrial problems, which may include simulation, the calculation of $\underline{g}$ can be very time consuming and is much longer than the overheads associated with calculating the search direction $\underline{p}$ given $\underline{g}$. The calculation of $\underline{g}$ is therefore the part of the algorithm where most benefit from parallelism can be expected to occur on such problems. The components of $\underline{g}$ are not identical, even for a simple function. If we can obtain an analytic expression for $\underline{g}$ the components can be computed in parallel on MIMD machines but not on SIMD machines. For an MIMD machine the parallel tasks are the calculation of the components of $\underline{g}$.

For most industrial optimisation problems, obtaining the analytic expression for $\underline{g}$ can be a very tedious process. We can use estimated values of g, obtained by some finite differences scheme, and this would be ideal for SIMD machines. There are efficient sequential variable metric and conjugate gradient codes which use a central difference scheme to approximate $\underline{g}$, so this may be expected to be acceptable on a parallel machine.

Let us consider the implementation of a modified Newton algorithm on the ICL DAP. On a sequential system the modified Newton techniques for solving unconstrained optimisation problems is one of the most efficient available for small dimensional problems where $2 < n \leq 5$. By introducing parallelism into Modified Newton technique, suitable for implementation on the ICL DAP, we would expect to increase the range for which Modified Newton technique is most efficient.

The Modified Newton approach is a second derivative method, i.e. not only do we need

to calculate g, but we also need to calculate the Hessian G. The elements of G are given by

$$G_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} .$$

We could approximate $\underline{g}$ and G by some finite differences scheme and this would be an ideal parallel calculation on the DAP.

The overheads in solving

$$G\underline{p} = -\underline{g}$$

to determine the search direction $\underline{p}$ will also be reduced on the DAP.

The final stage in the modified Newton algorithm is to select the step size α. This is often done by a search along a preselected curve. For the DAP implementation we consider three possible searches.

a) '1 - D' search (line search)
b) '2 - D' search)
c) '4 - D' search) appropriate choice of planes.

The '4 - D' search seems to be the most attractive proposition. For problems with $4 < n \leq 64$, the parallel algorithms based on this idea will be considerably more efficient and also reduce the number of Newton iterations. In the next section we describe the parallel Modified Newton algorithm which we implemented on the DAP.

Parallel (SIMD) Modified Newton algorithm

The parallel Modified Newton algorithm consists of the following steps:

Step 1: Initial guess $\underline{x}^{(o)}$; h, the step length and ε, the tolerance.

Step 2(a): Calculate $f(\underline{x} \pm ha_i \pm ha_j)$ all $j > i$, where a_i is a unit vector along the i^{th} axis. This function evaluation is a parallel calculation.

Step 2(b): Calculate the gradient vector $\underline{g}$ and the Hessian matrix G by the following finite differences scheme:

$$g_i = [f(\underline{x} + ha_i) - f(\underline{x} - ha_i)]/2h$$

$$G_{ij} = [f(\underline{x} + ha_i + ha_j) - f(\underline{x} + ha_i) - f(\underline{x} + ha_j) + f(\underline{x})]/h^2 \qquad i \neq j$$

$$G_{ii} = [f(\underline{x} + ha_i) - 2f(\underline{x}) + f(\underline{x} - ha_i)]/h^2$$

Step 3: Stop if $\max | g_i | < \varepsilon$

Step 4: Solve a set of linear simultaneous equations:

$G\underline{p} = -\underline{g}$, using DAP library subroutine FØ4GJNLE64.

Step 5: We considered three possible searches

(i) 4-dimensional grid search

The iterative step given by

$$X^{(k+1)} = \underline{x}^{(k)} + (A1)\alpha_1 \underline{p}^{(k)} + (A2)\alpha_2 \underline{g}^{(k)} + (A3)\alpha_3 d_3^{(k)} + (A4)\alpha_4 d_4^{(k)}$$

generates 4096 points

where $\alpha_1 = \alpha_1^{\text{initial}}$

$$\alpha_2 = \alpha_1 * \frac{g^T g}{g^T G g}$$

$$\alpha_3 = -\alpha_1 * \frac{d_3^T g}{d_3^T G d_3} \qquad (4.1)$$

$$\alpha_4 = -\alpha_1 * \frac{d_4^T g}{d_4^T G d_4}$$

$$\underline{d}_3 = \underline{x}^{(k)} - \underline{x}^{(k-1)}$$

$$\underline{d}_4 = \underline{x}^{(k-1)} - \underline{x}^{(k-2)}$$

A1, A2, A3 and A4 are (64 x 64) matrices given by

$A1_{\text{ith row}}$ = [-2,-2,-2,-2,-2,-2,-2,-2,:-1,-1,-1,-1,-1,-1,-1,-1:
0, ...0:1, ... 1:2, ... 2:3, ... 3:4, ... 4:
5, ... 5] (4.2)

$A2_{\text{ith row}}$ = [-2,-1,0,1,2,3,4,5:-2,-1,0,1,2,3,4,5: .....
-2,-1,0,1,2,3,4,5] (4.3)

$A3_{\text{ith col}}$ = [-3,-3,-3,-3,-3,-3,-3,-3:-2,-2,-2,-2,-2,-2,-2,-2:
1, ... 1:0, ... 0:1, ...1:2, ... 2:3, ... 2:
3, ... 3:4, ... 4] (4.4)

$A4_{\text{ith col}}$ = [-3,-2,-1,0,1,2,3,4:-3,-2,-1,0,1,2,3,4: ..... :
-3,-2,-1,0,1,2,3,4] (4.5)

The 4096 points generated are $^i x$, i=1.2. ... 4096

if $\min_{i=1,\ldots 4096} f(^i x^{(k+1)}) = f(\underline{x}^{(k)})$

then set $\alpha_1 = \frac{\alpha_1}{10}$ and recompute search step.

As soon as

$$\min_{i=1,\ldots 4096} f(^i x^{(k+1)}) < f(\underline{x}^{(k)})$$

then reset $\alpha_1 = \alpha_1^{\text{initial}}$

(ii) 2-dimensional grid search

The iterative step is

$$x^{(k+1)} = \underline{x}^{(k)} + (A1)\alpha_1 \underline{p}^{(k)} + (A2)\alpha_2 \underline{g}^{(k)}$$

where matrices A1 and A2 are given by:

(a) (4.2) and (4.3) respectively. (We call this VERSION A).

(b) $A1_{\text{ith col}}$ = [-16,-15.5,-15,-14.5,-14, ... 14.5,15,15.5]

$A2_{\text{ith row}}$ = [-16,-15.5,-15,-14.5,-14, ... 14.5,15,15.5]

(We call this VERSION B)

The iterative step will generate 4096 points; in fact, only 64 points are distinct for VERSION A, the rest are identical. The α's are given by (4.1).

For the 2-dimensional grid search, we considered the plane defined by $\underline{p}^{(k)}$ and $\underline{g}^{(k)}$, as this plane contains the directions usually chosen in sequential codes to make progress when G is nearly singular.

(iii) 1-dimensional search

The iterative step is

$$x^{(k+1)} = \underline{x}^{(k)} + (A1)\alpha_1 \underline{p}^{(k)}$$

where the matrix A1 is given by

(a) (4.2) (VERSION A)

(b) A1 has the values -102(0.05)102.75 in long vector order (VERSION B).

The iterative step will generate 4096 points in Version B but only 8 distinct points for VERSION A.

Step 6: $\underline{x}^{(k+1)} = \text{Arg} \min_{i=1,\ldots 4096} f(^i x^{(k+1)})$

- minimising over a preselected grid.

Step 7: Return to Step 2 with k = k+1.

The strong feature of the parallel algorithm is the parallel function evaluation in Steps 2 to 6. The DAP can perform 4096 function evaluations in parallel. It would be naive to assume that since the DAP performs 4096 function evaluations in parallel, the processing time would be of the order 4096 faster than the sequential calculation. In fact the arithmetic operations in the processing elements of the DAP are slower than the fast sequential machines (CDC 7600) due to the bit serial nature of each of the 4096 processing elements in the DAP. Let the ratios of the speed be τ. To get a rough idea of the value of τ, we obtained processing times for 4096 function evaluations of five 64-dimensional test problems (specified in Appendix A) on ICL DAP, ICL 2980 and DEC 1091. The processing times are displayed in Table 4.1.

Functions	Parallel evaluations DAP	Sequential evaluations DEC 1091	Sequential evaluations ICL 2980
Quadratic	0.045832	2.601	1.00928
Rosenbrock	0.109032	3.963	1.315296
Powell	0.067200	4.453	1.563352
Box (M)	11.62174	189.779	93.995920
Trignometric	0.351760	51.267	15.554040
Average ratio	1	20.7	9.3

Table 4.1. The processing times for 4096 function evaluations (time in seconds)

Experimental Results

Performance measurement.

The criteria we have used for measuring the performance of the parallel algorithm is to compare the processing times obtained on the DAP in relation to a sequential system. The sequential system we used was the DEC 1091 at The Hatfield Polytechnic. The codes used for the sequential system were:

(a) The standard Newton-Raphson method from the NAG Library, routine EØ4EBF [Ref. 12]. This is the nearest equivalent sequential algorithm.

(b) Variable Metric algorithm, the usually recommended approach on a sequential system for a 64 dimensional problem. We selected the Numerical Optimisation Centre's OPTIMA Library program OPVM [Ref. 13], an implementation of the Broyden-Fletcher-Shanno variable metric algorithm.

(c) Noticing that the structure of some of the test functions was symmetric and noting that this symmetry would favour a conjugate gradient approach, the Harwell Library routine VA14A [Ref. 14] was also used.

Numerical Results

The five test problems (specified in Appendix A) were run on the

i) DAP, using the parallel algorithm

ii) DEC 1091, using

(a) Modified Newton-Raphson, NAG routine, EØ4EBF

(b) Variable Metric, NOC OPVM

(c) Conjugate Gradient, Harwell VA14A.

We use approximate values of $\underline{g}$, the gradient vector and G, the Hessian, obtained using finite difference schemes described earlier in the section. For each test problem we used two sets of starting points, a symmetric and a nonsymmetric set. The numerical results are displayed in Tables A1 to A8.

Table A1 displays the processing times for the sequential codes.

Table A2 displays the DAP processing times for '1-D' search (VERSION A).

Table A3 displays the DAP processing times for '2-D' search (VERSION B).

Table A4 displays the DAP processing times for '2-D' search (VERSION A).

Table A5 displays the DAP processing times for '2-D' search (VERSION B).

Table A6 displays the DAP processing times for '4-D' search.

The performance ratio is displayed in Tables A7 and A8.

It will be seen from the numerical results that the parallel algorithm consistently outperformed the Newton-Raphson and Variable Metric sequential algorithms. In fact the DAP performed extremely well compared with the sequential Newton-Raphson algorithm.

Note of caution: Although the sequential Newton-Raphson is the nearest equivalent sequential algorithm to the parallel one, it is rarely used to solve a 64-dimensional problem. It is therefore fairer to use the Variable Metric method in the comparison.

For problems with special symmetry in the objective function, the parallel algorithm only just performs better than the Conjugate Gradient sequential algorithm (in terms of performance ratio); but for the nonsymmetric test function (Trignometric function) the parallel algorithm performed considerably better than the sequential Conjugate Gradient algorithm. Consider the '1-D' search algorithm: VERSION A performed better than VERSION B. For the '2-D' search algorithm, VERSION B performed better than VERSION A. Looking at the performance ratio and the number of iterations, there is not much to choose between the '2-D' search and the '4-D' search; for some problems the '2-D' search performed better than the '4-D' search and for other problems the '4-D' search performed better than the '2-D' search. This indicates that considerable further research is needed into the choice of the extra directions if the possible benefits of the 4-D search are to be obtained.

4.2 : Large dimensional problems

For large dimensional problems, say n > 120, the parallel Modified Newton approach would not be practicable, due to shortage of store on the DAP.

A different approach has been implemented on the ICL DAP for solving optimisation problems formulated by solving partial differential equations by finite elements as it was felt these would typify this class. The solution method follows the finite element approach and each processor of the DAP handles its own finite element. The solution is then completed by implementing linear or nonlinear versions of the conjugate gradient method as appropriate.

Consider the solution of partial differential equations

$$\text{(a)}\quad \frac{\partial}{\partial x}\, K_x \frac{\partial T}{\partial x} + \frac{\partial}{\partial y}\, K_y \frac{\partial T}{\partial y} = -Q \qquad (5.1)$$

$$\text{(b)}\quad \frac{\partial^2 y}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - Ru \frac{\partial u}{\partial x} = 0. \qquad (5.2)$$

It is wellknown that both the equations can be solved by minimising a functional

$$I = \iint F.dv.$$

The problem

We will briefly describe a solution method for solving the 2-D heat conduction equation (5.1). The solution to equation (5.1) will occur at the minimum of

$$I = \iint K_x \left(\frac{\partial T}{\partial x}\right)^2 + K_y \left(\frac{\partial T}{\partial y}\right)^2 - 2QT \qquad dv \qquad (5.3)$$

The objective function becomes one of solving

$$\min I = \int F(T(x)) \quad dV \qquad (5.4)$$

Subject to the appropriate boundary conditions, Newman type boundary conditions were imposed. Now when using finite elements the domain V is covered with a set of grid

points x_k and then divided into a set of elements V_i (of rectangular shape) with grid points at intersections. For each element there are shape functions ϕ_{ki} such that

$$\phi_{ki}(x) = 1 \quad \text{at} \quad x_k$$

$$\phi_{ki}(x) = 0 \quad \text{at} \quad x_j \text{ for } j \neq k \text{ and at all points } x \notin V_i$$

i.e. $\phi_{ki}(x) = (1 - \frac{x_k \pm x_i}{h})(1 - \frac{y_k \pm y_i}{h})/4.$

It is now possible to approximate T(x) by a linear combination of the shape functions.

$$\sum_k T_k \phi_{ki}(x) \tag{5.5}$$

so that equation (5.4) becomes approximately

$$\min I = \int F(\sum_k T_k \phi_{ki}(x)) \, dV. \tag{5.6}$$

On substituting (5.5) into equation (5.3) we get

$$F(T) = \iint K_x \{\sum_i \frac{\partial \phi_i}{\partial x} T_i\}^2 + K_y \{\sum_i \frac{\partial \phi_i}{\partial y} T_i\}^2 - 2Q\Sigma\phi_i T_i \quad dV \tag{5.7}$$

and on differentiating this with respect to T_i to get ∇F_i

$$\nabla_i F(T) = \iint 2K_x \frac{\partial \phi_i}{\partial x} \{\sum_j \frac{\partial \phi_j}{\partial x} T_j\} + 2K_y \frac{\partial \phi_i}{\partial y} \{\sum_j \frac{\partial \phi_j}{\partial y} T_j\} - 2Q\Sigma\phi_i \quad dV. \tag{5.8}$$

Differentiating once more gives $\nabla^2 F_{ij}$

$$\nabla^2_{ij} F(T) = \iint 2K_x \frac{\partial \phi_i}{\partial x} \frac{\partial \phi_j}{\partial x} + 2K_y \frac{\partial \phi_i}{\partial y} \frac{\partial \phi_j}{\partial y} \quad dV.$$

When the Q_i's are point sources at the nodes they can be moved outside the integration and then equation (5.7) becomes

$$F(T) = \sum_{el} \iint_{el} K_x \{\sum_i \frac{\partial \phi_i}{\partial x} T_i\}^2 + K_y \{\sum_i \frac{\partial \phi_i}{\partial y} T_i\}^2 \quad dV - 2Q_i T_i \tag{5.7a}$$

equation (5.8) becomes

$$\nabla_i F(T)_{el} = \iint_{el} 2K_x \frac{\partial \phi_i}{\partial x} \{\sum_j \frac{\partial \phi_j}{\partial x} T_j\} + 2Ky \frac{\partial \phi_i}{\partial y} \{\sum_j \frac{\partial \phi_i}{\partial y} \{\sum_j \frac{\partial \phi_j}{\partial y} T_j\} \, dV - 2Q_i \tag{5.8a}$$

and equation (5.9) becomes

$$\nabla^2_{ij} F(T)_{el} = \iint_{el} 2K_x \frac{\partial \phi_i}{\partial x} \frac{\partial \phi_j}{\partial x} + 2K_y \frac{\partial \phi_i}{\partial x} \frac{\partial \phi_j}{\partial y} \, dV. \tag{5.9a}$$

The solution of problem (5.6) then becomes equivalent to the solution of the set of equations (5.7). If the partial differential equations are linear this is a set of linear simultaneous equations which could be written Au = f.

Solution of the set of equations

When solving equations of the form

$A\underline{u} = \underline{f}$

where A is a real N x N matrix

$\underline{u}$ an N unknown vector

$\underline{f}$ an N known vector

if N is large there may be problems with computer storage. In the finite element approach the matrix A will be held as

$$\sum_{e=1}^{E} A^e \quad \text{(namely, the sum of all the elements matrices).}$$

All the values in A^e willl be zero except for those occurring in rows/columns corresponding to variables in the e^{th} element. It is, therefore, possible to solve the above set of equations without assembling the original matrix A. We have used the element matrices individually to evaluate $A\underline{p}$ element by element, by taking each element matrix in turn and multiplying it with the correct elements of the vector $\underline{p}$, the answers being stored in the appropriate processors. This makes the use of the conjugate gradient method advantageous.

Conjugate Gradient method

The conjugate gradient method is an iterative method that converges to the true solution in a finite number of iterations assuming no rounding errors. The idea was initially presented by Hestenes and Stiefel [9] and subsequently modified for optimisation purposes by Fletcher and Reeves [8].

We have successfully implemented parallel conjugate gradient algorithms, linear and nonlinear cases, on the ICL DAP parallel processing computer. We will briefly describe sequential linear and nonlinear conjugate gradient algorithms and then show how the solution of the partial differential equation (5.1) has been mapped onto the ICL DAP. For a brief description of the ICL DAP and the language for implementation refer to Section 2.

The linear conjugate gradient algorithm

The basic linear algorithm used for solving the set of equation $A\underline{u} = \underline{f}$ is described below.

1. Evaluate $\nabla^2 F$ (A).
2. Initialise the right hand side vector $\underline{f}^{(o)}$ (it is set equal to the source/sink points and zero elsewhere).
3. Set $\underline{u}^{(1)} = \underline{f}^{(o)}$ and $\underline{p}^{(o)} = \underline{f}^{(o)}$ ($\underline{u}$ being the unknown and $\underline{p}$ the search direction).
4. Evaluate $\underline{w}^{(o)} = A\,\underline{p}^{(o)}$. (We can perform this operation without having to assemble the matrix A).
5. Evaluate $\underline{f}^{(1)} = \underline{f}^{(o)} - \underline{w}^{(o)}$.
6. If $\underline{f}^{(1)T} \cdot \underline{f}^{(1)} < \varepsilon$ then stop.
7. Set $\underline{p}^{(1)} = \underline{f}^{(1)}$ and k = 1.

8. Evaluate $\underline{w}^{(k)} = A.\underline{p}^{(k)}$.
9. If $\underline{p}^{(k)T}.\underline{w}^{(k)} < \varepsilon$ then stop.
10. Set $\alpha^{(k)} = \dfrac{\underline{f}^{(k)T}.\underline{f}^{(k)}}{\underline{p}^{(k)T}.A.\underline{p}^{(k)}}$.
11. Update the unknowns $\underline{u}^{(k+1)} = \underline{u}^{(k)} + \alpha^{(k)}\underline{p}^{(k)}$
    and $\underline{f}^{(k+1)} = \underline{f}^{(k)} - \alpha^{(k)}\underline{w}^{(k)}$.
12. Set $\beta^{(k)} = \dfrac{\underline{f}^{(k)T}.\underline{f}^{(k)}}{\underline{f}^{(k-1)T}.\underline{f}^{(k-1)}}$.
13. Update the search direction $\underline{p}^{(k+1)} = \underline{f}^{(k+1)} + \beta^{(k)}.\underline{p}^{(k)}$.
14. If $\underline{p}^{(k+1)T}\underline{p}^{(k+1)} < \varepsilon$ then stop.
15. Set k = k+1, go to Step 8.

Note than an upper bound on the number of iterations is also imposed as a termination criteria.

On the parallel processor we need to evaluate the following products:

i) $\underline{f}^T\underline{f}$
ii) $\underline{p}^T.A.\underline{p}$
iii) $A.\underline{p}$
iv) $\underline{p}^T\underline{p}$.

Each of these can be subdivided into contributions from separate elements performed in parallel on separate processors.

The nonlinear conjugate gradient algorithm

The basic nonlinear conjugate gradient due to Fletcher and Reeves [8]is described below. Initially we guess values for $u_i^{(1)}$, i=1,2, ... N and then evaluate $\nabla^2 F$ at $\underline{u}^{(1)}$.

1. Set k = 1.
2. Evaluate $\nabla F^{(k)}$.
3. If $\nabla F^{(k)T}.\nabla F^{(k)} < \varepsilon$ then stop.
4. If k=1
   then set $\underline{p}^{(k)} = -\nabla F^{(k)}$
   else set $\beta^{(k)} = \dfrac{\nabla F^{(k)T}.\nabla F^{(k)}}{\nabla F^{(k-1)T}.\nabla F^{(k-1)}}$
   and $\underline{p}^{(k)} = -\nabla F^{(k)} + \beta^{(k)}\underline{p}^{(k-1)}$.
5. Evaluate α as $\alpha^{(k)} = \arg\min (F + \alpha p)$.
6. Update the unknown $\underline{u}^{(k+1)} = \underline{u}^{(k)} + \alpha^{(k)}\underline{p}^{(k)}$.
7. Set k = k+1.
8. If $k \leq k_{max}$ then go to Step 2
   else stop.

Again in this case we need to calculate the following:

i) ∇F)
ii) $\underline{p}$)
iii) $\nabla F^T.\underline{p}$)
iv) $\nabla F^T.\nabla F$)

and these can be computed in parallel in a similar manner to that used for the linear case.

Test problems and results

The main problems come from Stone [11] who ran an n x n problem for n = 11,21,31.

He considered equation (5.1); in this equation Q is a point source; at one point on the boundary T is fixed and the corresponding component of g set to zero; at the remaining points on the boundary the solution should set the normal component of ∇T to zero. The distribution of K_x and K_y will be described later.

Point sources

The problems all had three point sources and two point sinks and these were located as below:

For 11 x 11 at (2,2), (2,10), (8,2), (6,6), (10,10)
For 21 x 21 at (2,2), (2,20), (14,3), (10,11), (20,20)
For 31 x 31 at (3,3), (3,27), (23,4), (14,15), (27,27)
For 41 x 41 at (6,6), (6,36), (28,6), (20,20), (36,36) and
for 64 x 64 at (9,9), (9,60), (54,9), (34,34), (60,60) with values of
1.0 0.5 0.6 -1.83 -0.27 respectively.

The boundary conditions were imposed by fixing the temperature at one point on the boundary to 1.0 and the corresponding value of the gradient to 0.0.

Note that in our nonlinear algorithm the values of Q were divided by 4 since each Q comes into 4 sets of equations, its overall effect must be Q and not 4Q.

Conductivities

Four different conductivity distributions were use in the tests. They were specified independently in the x and y directions and located at the centre of an element.

PROBLEM 1. The model problem with K_x and K_y equal to unity over the entire region.
PROBLEM 2. The generalised model problem with K_x and K_y both constant, but with K_x 100 times greater than K_y.
PROBLEM 3. The heterogeneous test problem, here the region is divided into a number of smaller regions.
PROBLEM 4. This was the same as for 3, except that in one of the regions, where K_x and K_y had been set to 1.0, the conductivities came via random numbers.
PROBLEM 5. For the nonlinear p.d.e. equation (5.2) the least squares formulation was

adopted. The variables were augmented by introducing

$$a = \frac{\partial u}{\partial x}, \quad b = \frac{\partial u}{\partial y}$$

and the objective function I minimised over the values of u, a and b at the nodes, where I is given by

$$I = \iint (a - \frac{\partial u}{\partial x})^2 + (b - \frac{\partial u}{\partial u})^2 + (\frac{\partial a}{\partial x} + \frac{\partial b}{\partial y} - Rua)^2 \, dV.$$

Numerical Results

The problems were run on the DAP using the two codes, linear and nonlinear, and for comparison purposes the corresponding sequential versions were run on the Hatfield Polytechnic DEC 1091. The linear results are shown graphically in Figure B1 where the grid size is on the x-axis and the LOG of time on the y-axis. (Note that the LOG of time had to be used because of the large range of times obtained - particularly for the sequential cases). In all but a few cases (which have been indicated) the tolerance for termination was 1E-6. A suitable upperbound to the interations was set at 2* no. of points.

It is apparent from the graph that the parallel implementations are considerably faster than the sequential ones, Problem 1 was taken as far as a 64 x 64 problem for the parallel case just to prove that it was possible within a reasonable time but this could not be loaded on our sequential computer. The curves/lines for the parallel case increase only slightly in comparison with the sequential runs whose times soon become large as the number of unknowns is increased or as the ill-conditioning gets worse.

The results of the nonlinear problem are shown in Figure B2 which emphasise even more the benefits in time that can be obtained by solving the problem on the DAP rather than the sequential machine. It will be noted that the sequential runs were so expensive that the comparison tests had to be discontinued at a dimension where the problem was small enough only to require an eighth of the DAP.

Conclusions

The two parallel implementations have exhibited the fact that the solution of p.d.e.'s using this method of approach is very suitable to the SIMD class of machines and in particular the DAP.

Almost certainly one of the main advantages of the DAP in this case was the fact that the processors are connected to their nearest four neighbours, via row and column data highways, and the powerful shift indexing facilities make good use of this. Thus we have a simple but an extremely effective means of communicating with other nodes and neighbouring elements.

This facility can be compared to the two sequential programs which need to keep, for each element, a separate record of the four neighbouring (local) nodes, which must be set up and then indexed correctly. This is an overhead for the sequential case, not

to mention the fact that it is a complication in the writing and checking of the code which is not present in the algorithms. From the DAP's point of view it makes no significant difference whether it is calculating just 4 element matrices (a 3 x 3 problem) or 3,969 element matrices (a 64 x 64 problem) as either most or none of the processors will be switched off (masked out).

The algorithms that were employed for these solutions are basic with no sophisticated improvements though many are known. In theory we should terminate with the correct solution in at most N iterations (where N is the number of unknowns) but this assumes we use exact arithmetic with no rounding errors, which in practice is not the case. Improvements to the basic algorithms can be made, see for instance Powell [10].

Other major improvements are wellknown such as preconditioning, the multigrid approach, etc., where the system of equations

$$A\underline{x} = \underline{y}$$

is transformed into a new system

$$\hat{A}\hat{\underline{x}} = \hat{\underline{y}}$$

which will have a much smaller condition number in order to speed up the convergence. Parallel implementation of such a method will be the basis of additional work.

4.3 : Multiextremal global optimisation

In this section we begin with a brief introduction in which we define the global optimisation problem, give a classification of methods and briefly describe the nature of one class of methods.

Introduction

The global optimisation problem is posed as follows:

Consider the problem

$$\min f(\underline{x}), \quad S \subset R^n$$

where $S = \{\underline{x}: \ U_i \leq x_i \leq l_i, \ i = 1,2, \ldots n\}$.

We shall assume that the objective function $f(\underline{x})$ is nonconvex and possesses more than one local minimum. The problem is to locate that local minimum point $\underline{x}^*$ with the least function value. For some problems we have more than one local minimum that is global.

The methods for solving global optimisation problems are classified as deterministic or probabilistic. It is generally accepted that deterministic methods can only be applied to a restricted subset of objective functions $f(\underline{x})$, whereas more general statements can sometimes be made for a probabilistic method. Hence for a general problem probabilistic methods are often preferred. In this paper we will only consider methods belonging to that classification. Probabilistic methods rely on the following result:

Assume that a finite number of points chosen at random are distributed uniformly over a finite region S. If A is a subset of S with a measure m such that

$$\frac{m(A)}{m(S)} \geq \alpha > 0$$

and p(A,N) is the probability that at least one point of a sequence of N random points lies in A then

$$\lim_{N\to\infty} p(A,N) = 1.$$

A description of some of the probabilistic methods can be found in Dixon & Szegö [16].

Although many alternative probabilistic methods have been suggested none could claim to be the 'best' as no agreed measure of performance is available. In practice very few real multiextremal nonconvex global optimisation problems have been solved. The reason for this is the high 'computational costs' involved when implementing the probabilistic algorithm on a sequential system. One of the main contributing factors for this is the time of repeated function evaluations, the more complicated the nature of the objective function the more these costs increase. The clustering type of probabilistic algorithms are among the most successful and these generally consist of three phases:

a) selection of N trial points chosen at random from the region of interest and the computation of the associated function values,

b) local search phase. A few iterations of a local minimisation routine are run from each of the trial points. This is done to push the points near to a local minimum,

c) clustering phase. Find clusters amongst the resulting points using clustering analysis techniques and to return to phase b with a reduced number of points.

The choice of N, the number of trial points, is quite critical. If it is too small, then the exploration of the region of interest will be insufficient and possible optimum locations may be overlooked. So the greater the value of N the more likelihood there is of the algorithm locating a global minimum. Using a parallel system, say, MIMD or SIMD in nature, we could choose large N (and perform function evaluations in parallel), thus increasing the probability of the algorithm locating a global minimum.

A number of sequential probabilistic algorithms have been suggested, e.g. Price [17], Torn [18] and Boender [19]. We will consider the Controlled Random search algorithm of Price, intended for a mini-computer, and describe how we implemented the parallel versions of it on an SIMD array system and an MIMD system. The exploitation of parallelism that we will be describing will be at the level of designing parallel algorithms. When designing a parallel algorithm we shall bear in mind the target architecture. First we will summarise the sequential algorithm and present the numerical results for ten test problems.

The sequential Price's CRS algorithm consists of the following steps:

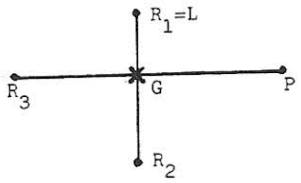
Step 1: A sample of N points from the domain of an objective function is taken. The points are uniformly distributed over the feasible region, S. We have chosen N = 10(n + 1). Evaluate the function at each point.

Step 2: Choose the point M which has the greatest function value, fM (worst point). Choose the point L which has the least function value, fL (best point).

Step 3: Choose n distinct points, R_2, R_3, ... R_{n+1}, at random from the set of N points. Let R_1 = L. Determine G, the centroid of the first n points R_1, R_2, ... R_n. The next trial point is defined to be

$$P = 2G - R_{n+1}.$$

External reflection of R_{n+1} about G (pictorial illustration below for n = 2)



Step 4: If the trial point P is within the feasible region S then proceed to Step 5; otherwise repeat Step 3.

Step 5: Determine f_p, the function value at the point P. If $f_p <$ fM, accept point P and proceed to Step 6; otherwise repeat Step 3.

Step 6: Replace the worst point M in the set of N points, by point P.

Step 7: If termination criterion is satsfied, stop; otherwise repeat from Step 2.

The termination criteria we have used is:

at the i^{th} iteration

if $|f_i - f_{i-1}| < \varepsilon$ and $|f_{i-1} - f_{i-2}| < \varepsilon$ then stop,

where ε is some tolerance.

Numerical Results

The sequential algorithm was coded in FORTRAN and run on the Hatfield Polytechnic DEC 1091. The test problems chosen are described in Appendix C. Computational results of tests on the standard test functions are collected in Table C1. The CRS procedure has the advantage of being simpler and requires less computer storage than the more sophisticated clustering algorithm.

Parallel (SIMD) Controlled Random search algorithm

Consider the interaction of the CRS algorithm with the SIMD ICL DAP. A parallel version of the CRS algorithm suggested by Dixon and Patel [15] and modified by Ducksbury is shown in Figure 4.3.

Numerical Results

The results of this parallel algorithm are displayed in Table C2 (sequential against parallel). It is observed that the parallel algorithm performed better than the sequential algorithm, in terms of function evaluation and performance ratio. The performance ratio ranges from 3.2 to 68. The lower ratio is for the test problem

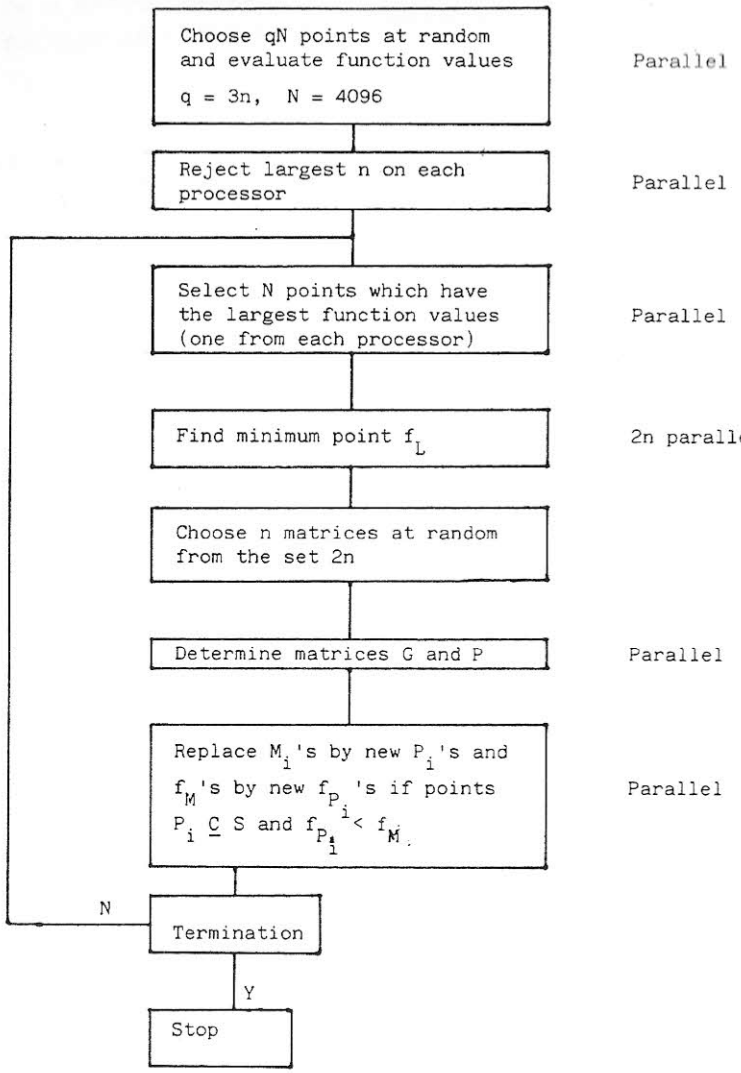


Figure 4.3

where there is only one global minima and the larger ratio is for the problems which possess multiple global minima. The algorithm is only suitable for small values of n, in the range $2 \leq n \leq 5$. This restriction was due to shortage of store on the DAP at the time these tests were undertaken. The store on the DAP has subsequently been increased.

Parallel (MIMD) Controlled Random Search algorithm

A parallel version of the Controlled Random Search algorithm for inplementation on the multiprocessor NEPTUNE system consists of the following steps:

Step 1: Choose N points at random in the region S to be searched, where N = 10(n+1).

Evaluate the function at each point. This is a sequential step, but it can be parallelised. A set of N points and their associated function values are stored in the global memory, i.e. accessible to all the processors.

Processor i(i=0,1,2,3) then executes the following:

Step 2: Processor i chooses $(i+1)^{st}$ worst point, M_i, with function value f_{M_i}.

$f_{M_o} = \max (f_j, j=1,2, \ldots N)$

$f_{M_k} = \max (f_j, j=1,2, \ldots N, j \neq M_t, t = 0,1, \ldots (k-1))$.

Choose the point L which has the least function value, f_L (best point)

$f_L = \min (f_j, j=1,2, \ldots N)$.

Steps 3 and 4: Generate a new point, P, and check for its feasibility, as in the sequential algorithm.

Step 5: Determine f_p, the function value at the point P. If $f_p < f_{M_i}$ accept point P and proceed to Step 6; otherwise repeat Step 3.

Step 6: Replace the $(i+1)^{st}$ worst point M_i in the set of N points stored in the global memory by the new point P.

Step 7: Terminate or repeat from Step 2, as in the sequential algorithm.

The content of the global memory is continually being modified by each of the processors and there is no communication between the processors. Hence the process is an asynchronous process.

Numerical Results

The parallel algorithm was run on the four processor NEPTUNE system at Loughborough University of Technology. The test problems chosen are described in Appendix C.

For each test problem, the parallel algorithm was first run on one processor (processor 0), then on two processors (different combinations of the processors, i.e. 0,1; 0,2; 0,3), then on three processors (0,1,2; 0,1,3; 0,2,3) and finally on all four processors. A number of runs for each test problem were carried out. The numerical results are displayed in Table C3. The performance of the parallel algorithm, in terms of speed-up and number of function evaluations, improves as we increase the number of processors. We observe that the algorithm converges faster (i.e. less function evaluations) as we increase the number of processors.

5. CONCLUSIONS

In this paper we have presented results indicating that in 3 very separate circumstances the use of parallel processing systems can greatly affect the computer time involved in solving optimisation problems. This subject appears a very fruitful area for future research.

6. ACKNOWLEDGEMENTS

The work in this report was carried out under two research grants from the SERC, nos.

GR/B/1794/5 and GR/B/4665/5. The DAP Support Unit at Queen Mary College, University of London are gratefully acknowledged for their permission to use the DAP and Loughborough University for their permission to use the NEPTUNE system.

REFERENCES

1] Rodrigue, G and Greenbaum, A, The incomplete Cholesky Conjugate Gradient method for the STAR, Lawrence Livermore Lab. Report, UCID-17574, 1977.

2] Rodrigue, G, Dubois, P F and Greenbaum, A, Approx. the inverse of a matrix for use in iterative algorithms on a vector processor, Computing 22, pp 257-268, 1979.

3] Schreiber, R S, The implementation of the conjugate gradient method on a vector computer, Stanford University, CA 94305, 1981.

4] Sargan, J D, Chong, Y Y and Smith, K A, Nonlinear Econometric Modelling on a Parallel Processor, DAP Support Unit, Queen Mary College, London, October 1981.

5] Sauber, W, A data flow architecture implementation, Technical Report, Texas Instruments Incorporated, Austin, Texas, 1980.

6] Brown, A H O, The development of computer optimisation procedures for use in aero engine design, in 'Optimisation in Action,' ed. L C W Dixon, Academic Press, 1976.

7] Dixon, L C W, Global optima without convexity, in 'Design and Implementation of Optimisation Software,' 1978.

8] Fletcher, R and Reeves, C M, Function minimisation by conjugate gradients, Computer Journal, Vol.7, 1964.

9] Hestenes, M and Stiefel, E, Methods of conjugate gradients for solving linear systems, Journal Res.Nat.Bu.Standards, No.49, 1952.

10] Powell, M J D, Restart procedures for the conjugate gradient method, Math. Programming, Vol.12, pp 221-154, 1975.

11] Stone, H L, Iterative solution of implicit approximations of multi-dimensional PDE's, SIAM Num.Anal., Vol.5, No.3, September 1968.

12] NAG Manuals (Mark 8).

13] OPTIMA Manual, Numerical Optimisation Centre, The Hatfield Polytechnic, Issue No.4, 1982.

14] Harwell Subroutine Library, 1981.

15] Dixon, L C W and Patel, K D, The place of parallel computation in numerical optimisation II: The multiextremal global optimisation problem, TR117, Numerical Optimisation Centre, Hatfield Polytechnic, 1981.

16] Dixon, L C W and Szegö, G P, The global optimisation problem: An introduction, pp 1-15, in 'Towards Global Optimisation 2," eds. Dixon and Szegö, North Holland Publishing Company, 1978.

17] Price, W L, A new version of the controlled random search procedure for global optimisation, TR, Enginerring Dept., University of Leicester, 1981.

18] Törn, A A, A search clustering approach to global optimisation, in 'Towards Global Optimisation 2,' eds. Dixon and Szegö, North Holland, 1978.

19] Boender, C G E, et al, A stochastic method for global optimisation, Math. Programming, 1983.

20] Evans, D J, Barlow, R H, Newman I A and Woodward, M C, A guide to using the NEPTUNE parallel processing system, Dept. of Computer Studies, Loughborough University of Technology, 1982.

21] DAP Manuals, Technical Publications, TP 6755, TP 6918, TP 6920, ICL, Putney, London SW15.

APPENDIX A

The following five 64-dimensional test problems were used:

a) Quadratic function

$$f(\underline{x}) = \sum_{i=1}^{32} 100x_{2i}^{2} + (1-x_{2i-1})^{2}.$$

b) Rosenbrock function

$$f(\underline{x}) = \sum_{i=1}^{32} 100(x_{2i-1}^{2} - x_{2i})^{2} + (1-x_{2i-1})^{2}.$$

c) Powell function

$$f(\underline{x}) = \sum_{i=1}^{16} [(x_{4i-3} + 10x_{4i-2})^{2} + 5(x_{4i-2} - x_{4i})^{2} + (x_{4i-2} - 2x_{4i-1})^{4} + 10(x_{4i-3} - x_{4i})^{4}].$$

d) Box (M) function

$$f(\underline{x}) = \sum_{j=1}^{32} \sum_{i=1}^{10} [\exp(-x_{2j-1}\, t_i) - 5\exp(-x_{2j}\, t_i) - \exp(-t_i) + 5\exp(-10t_i)]^{2}.$$

$$t_i = \frac{i}{10},\ i=1,2,\ \ldots\ 10.$$

e) Trigonometric function

$$f(\underline{x}) = \sum_{i=1}^{64} [64 + i - \sum_{j=1}^{64} (A_{ij} \operatorname{Sin}(x_j) + B_{ij} \operatorname{Cos}(x_j))]^{2}.$$

$$A_{ij} = \delta_{ij},\quad B_{ij} = i\,\delta_{ij} + 1.$$

$$\delta_{ij} = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j. \end{cases}$$

The first four problems have symmetric objective functions.

Table A1

Processing times for the sequential codes (time in seconds)

Function Starting Point $x^{(0)}$	Newton-Raphson CPU Time	Variable Metric			Conjugate Gradient		
		CPU Time	No. of Funct. Calls	No. of Grad. Calls	CPU Time	Iteration	No. of Funct. & Grad. Calls
Quadratic	15.00	1.80	8	5	1.23	3	9
$(0,1,\ldots.)^T$	16.60	2.10	12	7	1.18	4	9
non-symmetric							
Rosenbrock							
$(0,1,\ldots.)^T$	154.23	72.11	414	210	5.67	19	48
$(-1.2,1,\ldots.)^T$	145.97	103.35	683	292	8.36	27	72
non-symmetric	140.98	80.78	496	249	11.36	49	103
Powell							
$(3,-1,0,3\ldots.)^T$	112.47	53.16	298	142	5.91	22	49
non-symmetric	134.97	41.28	218	114	11.06	35	87
Trigonometric							
$(1/64,1/64,\ldots)^T$	F	66.18	74	43	37.56	14	27
non-symmetric	2604.31	286.65	455	191	78.68	24	52
Box(M)	4181.86	1161.27	429	227	137.81	6	16
$(1,2,1,2,\ldots)^T$	7196.09	3194.39	1263	621	354.02	19	40
non-symmetric							

Table A2

DAP processing times for '1-D' search [VERSION A]

Function	Starting point $\underline{x}^{(0)}$	initial α_1	h	ϵ	Iter.	DAP time (secs.)
Quadratic	$(0,1,\ldots.)^T$ non-symmetric	1.0 1.0	0.1 0.1	0.0001 0.0001	2 2	0.8919624 0.803285
Rosenbrock	$(0,1,0,1,\ldots)^T$	0.5	0.001	0.001	14	5.532199
	$(-1.2,1.0,\ldots)^T$ non-symmetric	0.5 1.0	0.001 0.001	0.001 0.001	20 26	8.258047 9.842461
Powell	$(3,-1,0,3,\ldots.)^T$ non-symmetric	1.0 1.0	0.01 0.01	0.0001 0.0001	5 3	1.746016 1.248776
Trigonometric	$(1/64,\ldots.)^T$ non-symmetric	1.0 0.5	0.001 0.001	0.001 0.001	12 14	9.647102 13.33931
Box (M)	$(1,2,1,2,\ldots.)^T$ non-symmetric	0.5 0.125	0.01 0.01	0.0001 0.0001	18 21	452.6223 508.0827

Table A3

DAP Processing times for '1-D' search [VERSION B]

Function	Starting point $\underline{x}^{(0)}$	initial α_1	h	ϵ	Iter.	DAP time (secs.)
Quadratic	$(0,1,\ldots.)^T$ non-symmetric	0.5 0.5	0.1 0.1	0.001 0.001	2 2	0.79532 0.774928
Rosenbrock	$(0,1,0,1,\ldots.)^T$	0.5	0.001	0.001	8	3.60468
	$(-1,2,1.0,\ldots)^T$ non-symmetric	0.5 0.125	0.001 0.001	0.001 0.001	25 34	9.791973 14.340800
Powell	$(3,-1,0,3,\ldots.)^T$ non-symmetric	1.0 1.0	0.01 0.01	0.001 0.001	16 10	5.338949 3.542872
Trigonometric	$(1/64,\ldots.)^T$ non-symmetric	1.0 0.5	0.001 0.001	0.001 0.001	13 14	11.07125 13.379770
*Box (M)	$(1,2,1,2,\ldots.)^T$ non-symmetric	- -	0.001 0.001	0.001 0.001	- -	F F

* overflow in function evaluation

Table A4

DAP processing times for '2-D' search [VERSION A]

Function	Starting point $\underline{x}^{(0)}$	initial α_1	h	ϵ	Iter.	DAP time (secs.)
Quadratic	$(0,1,.....)^T$	1.0	0.1	0.0001	2	0.985168
	non-symmetric	1.0	0.1	0.0001	2	0.966224
Rosenbrock	$(0,1,....)^T$	0.5	0.001	0.001	12	6.486191
	$(-1.2,1.0,...)^T$	0.5	0.001	0.001	11	5.86484
	non-symmetric	0.5	0.001	0.001	71	36.626720
Powell	$(3,-1,0,3,...)^T$	1.0	0.01	0.0001	4	1.929656
	non-symmetric	1.0	0.01	0.0001	3	1.500664
Trigonometric	$(1/64,....)^T$	0.5	0.001	0.001	11	11.10786
	non-symmetric	0.5	0.001	0.001	17	17.57672
Box (M)	$(1,2,1,2,...)^T$	0.25	0.01	0.001	5	130.9869
	non-symmetric	0.25	0.01	0.001	8	202.3719

Table A5

DAP processing times for '2-D' search [VERSION B]

Function	Starting point $\underline{x}^{(0)}$	initial α_1	h	ϵ	Iter.	DAP time (secs.)
Quadratic	$(0,1,.....)^T$	1.0	0.1	0.001	2	0.968968
	non-symmetric	1.0	0.1	0.001	2	0.949536
Rosenbrock	$(0,1,....)^T$	1.0	0.001	0.001	6	3.282704
	$(-1.2,1.0,...)^T$	1.0	0.001	0.001	4	2.290592
	non-symmetric	0.5	0.001	0.001	25	12.886690
Powell	$(3,-1,0,3,...)^T$	1.0	0.001	0.001	7	3.154320
	non-symmetric	1.0	0.001	0.001	7	3.164520
Trigonometric	$(1/64,....)^T$	0.5	0.001	0.001	10	10.048290
	non-symmetric	0.5	0.001	0.001	15	15.523480
Box (M)	$(1,2,1,2,...)^T$	0.25	0.01	0.001	4	107.820100
	non-symmetric	0.0625	0.01	0.001	10	251.929100

Table A6

DAP processing times for '4-D' search

Function	Starting point $\underline{x}^{(0)}$	initial α_1	h	ϵ	Iter.	DAP time (secs)
Quadratic	$(0,1,....)^T$	1.0	0.1	0.0001	2	1.237488
	non-symmetric	1.0	0.1	0.0001	2	1.210280
Rosenbrock	$(0,1,....)^T$	1.0	0.001	0.001	6	4.279285
	$(-1.2,1.0,...)^T$	1.0	0.001	0.001	7	4.949902
	non-symmetric	0.5	0.001	0.001	41	28.211770
Powell	$(3,-1,0,3,....)^T$	1.0	0.01	0.0001	4	2.52372
	non-symmetric	1.0	0.01	0.0001	3	1.920984
Trigonometric	$(1/64,....)^T$	1.0	0.001	.0.001	12	13.998930
	non-symmetric	0.5	0.001	0.001	7	8.657191
Box (M)	$(1,2,....)^T$	0.5	0.01	0.001	4	107.9466
	non-symmetric	0.125	0.01	0.001	13	323.9529

Table A7

Performance measurement [VERSION A]

Define 'speed-up' ratio, = $\frac{\text{processing time using a sequential system}}{\text{processing time using the DAP}}$

Function	Starting point $\underline{x}^{(0)}$	Newton-Raphson DAP			Variable-Metric DAP			Conjugate-Gradient DAP		
		1-D	2-D	4-D	1-D	2-D	4-D	1-D	2-D	4-D
Quadratic	$(0,1,....)^T$	16.8	15.2	12.1	2.0	1.8	1.4	1.4	1.2	0.99
	non-symmetric	20.7	17.2	13.7	2.6	2.2	1.7	1.5	1.2	0.97
Rosenbrock	$(0,1,...)^T$	27.9	23.8	36.0	13.0	11.1	16.8	1.0	0.9	1.3
	$(-1.2,1.0,...)^T$	17.7	24.9	29.5	12.5	17.6	20.9	1.0	1.4	1.7
	non-symmetric	14.3	3.6	5.0	8.2	2.2	2.9	1.1	0.3	0.4
Powell	$(3,-1,0,3,...)^T$	64.4	58.3	44.6	30.4	27.6	21.1	3.4	3.1	2.3
	non-symmetric	108.1	89.9	70.3	33.1	27.5	21.5	8.9	7.4	5.8
Trigonometric	$(1/64,....)^T$	-	-	-	6.9	5.9	4.7	3.9	3.4	2.7
	non-symmetric	195.2	148.2	300.8	21.5	16.3	33.1	5.9	4.5	9.1
Box (M)	$(1,2,....)^T$	9.8	31.9	38.7	2.7	8.9	10.8	0.3	1.1	1.3
	non-symmetric	14.2	35.6	22.2	6.3	15.9	9.9	0.7	1.7	1.1

Table A8

Performance measurement [VERSION B]

Define 'speed-up' ratio, = $\frac{\text{processing time using a sequential system}}{\text{processing time using the DAP}}$

Function	Starting point $\underline{x}^{(0)}$	Newton-Raphson DAP		Variable Metric DAP		Conjugate gradient DAP	
		1-D	2-D	1-D	2-D	1-D	2-D
Quadratic	$(0,1,....)^T$	18.9	15.5	2.3	1.9	1.5	1.3
	non-symmetric	21.4	17.5	2.7	2.2	1.5	1.2
Rosenbrock	$(0,1,...)^T$	42.8	47.0	20.0	22.0	1.6	1.7
	$(-1.2,1.0,...)^T$	14.9	63.7	10.6	45.1	0.9	3.6
	non-symmetric	9.8	10.9	5.6	6.3	0.8	0.9
Powell	$(3,-1,0,3,...)^T$	21.1	35.7	10.0	16.9	1.1	1.9
	non-symmetric	38.1	42.7	11.7	13.0	3.1	3.5
Trigonometric	$(1/64,....)^T$	-	-	6.0	6.6	3.4	3.7
	non-symmetric	194.6	167.8	21.4	18.5	5.9	5.1
Box (M)	$(1,2,....)^T$	-	38.8	-	10.8	-	1.3
	non-symmetric	-	28.6	-	12.7	-	1.4

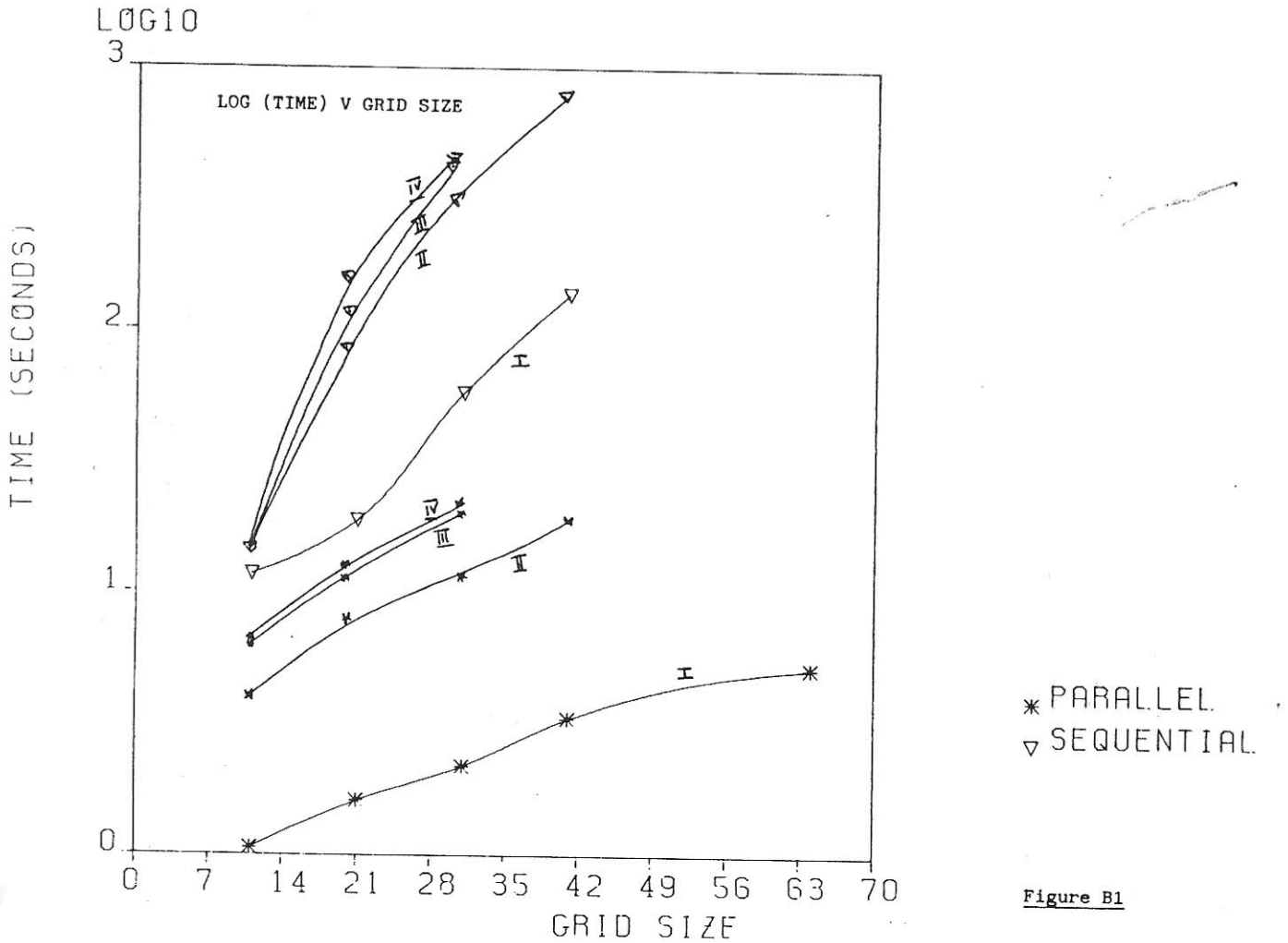


Figure B1

Table C1

Numerical results for the sequential CRS algorithm

Number of global points = 10(n+1)

Tolerance, ϵ = 0.000001

function	No. of function evaluations	x*	f(x*)
Goldstein and Price	792	(0, -1.0)	3.0
Branin	744	(-3.14163, 12.27467)	0.39789
	410	(3.13630, 2.26562)	0.39820
		(9.46559, 2.45409)	0.40897
Hartman's family			
m=4, n=3	1276	(0.11514, 0.55568, 0.85253)	-3.86278
m=9, n=6	3804	(0.40474, 0.88258, 0.84349, 0.13724, 0.03849)	-3.20316
m=4, n=6	6416 ϵ=0.0000001	(0.20168, 0.15004, 0.47699, 0.27515, 0.31168, 0.65732)	-3.32237
Shekel's family			
n=4, m=5	3996	(4.0, 4.0, 4.0, 4.0)	-10.15320
n=4, m=7	2872	(4.00069, 4.00080, 3.99939, 3.99957)	-10.40294
n=4, m=10	3284	(4.00066, 4.00058, 3.99965, 3.99954)	-10.53641
3 Hump Camel-back	706	(0.00000, 0.00044)	0.00000
6 Hump Camel-back	676	(-0.09002, -0.71249)	-1.03163
		(0.08999, 0.71341)	-1.03162
Treccani	412	(-0.00043, -0.00009)	0.00000

Table C2

Numerical results for parallel (SIMD) CRS algorithm

Function	Sequential		Parallel (SIMD)		Performance ratio
	Time(s)	fn.evals.	Time(s)	fn.evals.	
Six Hump Camel-back (2 global minima)	0.36	288	0.093	7 Note (1)	12.5
	0.81	798			
	1.17	1086			
Branin (3 global minima)	0.47	360	0.118	8 Note (1)	26.8
	0.97	654			
	1.73	1024			
	3.17	2038			
Goldstein and Price (1 global minimum)	0.45	481	0.13	8	3.5
Hartman's n=3 (1 global minimum)	0.88	503	0.276	11	3.2
Shekel Note (3)					
m=5	3.89	2732	0.453	14	8.6
m=7	3.82	2423	0.55	14	6.9
m=10	4.31	2567	0.693	14	6.2
(1 global minimum)					
Shubert 2-D (18 global minima)	average 1.58/min	average 995	0.233 Note (2)	8	68
	total 15.8				
	Note (2)				

Notes:

(1) Multiple global minima are identified in one run.

(2) The parallel algorithm picked up 11 minima in one run hence the same 11 minima from the sequential version were used for the comparison (these 11 being picked up in 10 runs).

(3) Located the approximate global minimum.

Table C3

Numerical results for parallel (MIMD) CRS algorithm

function	No. of Processors	Speed-up range	fn.evals. for the best run			
			Proc. 0	Proc. 1	Proc. 2	Proc. 3
3 Hump Camel-back	1		432			
	2	1.21 - 2.45	181	154		
	3	1.53 - 2.64	180	149	158	
	4	2.01 - 2.65	185	152	154	159
6 Hump Camel-back	1		434			
	2	1.29 - 1.86	232	212		
	3	1.80 - 1.99	234	193	212	
	4	2.05 - 3.21	170	128	128	129
Treccani	1		458			
	2	1.46 - 1.86	264	262		
	3	2.11 - 2.27	233	215	196	
	4	2.48 - 2.78	148	138	133	138
Goldstein and Price	1		547			
	2	1.38 - 2.11	303	271		
	3	1.96 - 2.75	212	182	185	
	4	2.11 - 3.01	224	189	186	210
Branin	1		475			
	2	1.27 - 1.69	257	221		
	3	1.81 - 2.39	205	170	180	
	4	2.37 - 3.74	139	105	107	120
Shekel's family						
n=4, m=5	1		3746			
	2	1.43	2388	2355		
	3	1.91	1818	1771	1751	
	4	2.76	1073	1014	1065	1101
n=9, m=7	1		2932			
	2	1.51	1962	1969		
	3	1.83	1547	1546	1542	
	4	2.23	1290	1259	1234	1286
n=4, m=10	1		2836			
	2	1.45	1941	1901		
	3	1.82	1602	1611	1609	
	4	2.41	1170	1146	1191	1183
Hartman's family						
m=4, n=3	1		1022			
	2	1.29 - 1.83	557	516		
	3	2.15 - 2.49	389	368	371	
	4	2.98 - 3.80	271	238	253	244
m=4, n=6	1		3348			
	2	-	-			
	3	2.17 - 2.37	1370	1325	1369	
	4	2.39 - 2.91	1154	1095	1116	1141