Exploiting Vector Computers by Replication

PETER R. BENYON
CSIRO Division of Computing Research, GPO Box 1800, Canberra, ACT 2601, Australia

Two broad classes of problem suitable for computers with vector architecture can be distinguished. The first have a systematic structure that makes them natural candidates for this type of machine. The second lack such structure but the algorithm as a whole has to be run many times. It may then be possible to vectorise across a large number of conceptual replicates of the same algorithm rather than trying to vectorise within the one algorithm. The conditions allowing this are stated. An advantage is that even scalars in the original become long vectors in the replicated version. A disadvantage is the amount of storage needed. The application of replication to a continuous simulation problem is discussed.

1. INTRODUCTION

Computers having pipelined vector architecture, such as the Cyber 205, are generally considered to be rather specialised machines particularly suitable for solving multi-dimensional partial differential equations. However, it is being found that there are other classes of problem that can be made to vectorise well, significantly widening the applications for which such machines are attractive. One such class of problem is identified here for which it is shown that a method termed 'replication' can be very effective. The basic idea is not new, but most discussions of it lie buried in the specialised literature related to particular applicatons.[1-8] The aim here is to bring out the generality of the approach and make it more widely known.

To get the best out of a vector machine it is necessary to organise the calculations so that they consist as much as possible of similar operations on all elements of long vectors. To qualify as a 'vector' in the case of the Cyber 205, elements must be stored in consecutive locations. Achieving a high proportion of vectorisation is very important, since if only 90% of the work can be vectorised, the speed may be only 10% of that for full vectorisation. This is because vector arithmetic is so much faster than scalar arithmetic on machines of this class. Replication is one technique aimed at achieving a high degree of vectorisation.

Section 2 will describe the general principle and Section 3 some experiences in trying it out in a particular case.

2. GENERAL METHOD

2.1 Problems requiring multiple independent runs

The type of problem to be considered can be anything that requires a substantial algorithm to be executed many times over, but where the different executions are independent of each other.

Two commonly occurring reasons for multiple execution are:

(1) the presence of random numbers in the calculation, with the attendant need for large batches of runs to obtain means and other statistics; and

(2) the requirement to vary parameters in search of an optimum, leading to large numbers of runs using different trial sets of parameters.

In the first case, the different runs are independent of each other, but in the second the usual 'hill climbing' methods do not as they stand satisfy the requirement of independence, since the next set of parameters to be tried depends upon the results obtained with earlier sets. However, once the designers of these search algorithms realise they can have 1000 independent runs (i.e. evaluations of the objective function) for the price of about 20 sequential ones, they will doubtless think of ways to exploit that capability.

In any case, search algorithms cannot be guaranteed to find the global optimum; so to increase the chances that the optimum found is not just local, one often repeats the whole search a number of times, each time starting from a different point in the parameter space. A large number of searches starting from different points could all be carried out as a batch, and these different searches would be independent.

The two reasons for multiple execution sometimes occur together. Many different sets of parameters may have to be tried for optimisation purposes, while for each set many runs may be needed to obtain the average result in the presence of random variations.

2.2 Vectorised multiple runs

Given some algorithm to be executed many (say 1000) times, imagine we had 1000 computers side by side. We could load the same program into all of them, each with its own set of parameter values or its own starting number for the random number generator, and set them all running simultaneously. In general they would not remain in step because the path taken through a computer algorithm is not usually the same every time it is executed, due to different numerical values influencing the outcome of IF and similar tests. However, suppose we could force them to remain in step. As we look across the line of 1000 computers, each variable or array element in the program would appear as a vector of 1000 numbers with each machine working on a different element of the vector. Clearly, on a vector machine we could emulate those 1000 simultaneous executions using vectors 1000 long. This approach of vectorising across many replicates of a complete algorithm, rather than trying to vectorise within the algorithm, will be called 'vectorising by replication'. When it can be used, it is very effective, since even scalars in the original algorithm become long vectors.

This method requires two conditions to be met: (1) the different replicates must be independent of each other;

and (2) they must remain in step with each other. If both of these conditions had to be satisfied strictly there would be few cases where replication could be used, but it is this writer's belief that a great many applications can be forced to meet these conditions without too much effort. To illustrate, a common reason for violating the in-step condition is likely to be that some process has to be iterated to convergence. In some runs it may take only three iterations to converge, while in others it may take as many as ten. One would simply have to force all runs to grind around for ten iterations anyway. In other words, one must be prepared to sacrifice efficiency in the small to gain much greater efficiency in the large.

2.3 Applications

The technique has appeared in a variety of guises and under a variety of names in different fields of application. Saunders, who has applied it in quantum chemistry,[6] aptly calls it 'extrinsic vectorisation' as distinct from the 'intrinsic vectorisation' that may be possible within the algorithm. Hegarty,[4] also in connection with quantum chemistry, uses the term 'data driven', as opposed to 'algorithm driven'. Temperton[8] has applied the method in signal processing, where a large number of fast Fourier transforms can be computed at once.

So far, though, interest in replication has been less than might have been expected. This is probably because people have not realised how often it will be possible to force satisfaction of the necessary conditions.

One limitation to using the idea in some cases could be the amount of memory needed. However, vector methods in general tend to rely on plentiful memory. Fortunately very large amounts are becoming available on this class of machine.

Continuous simulation is one field where multiple runs are often needed and where it should generally be fairly easy to satisfy the in-step requirement, provided a fixed-step rather than variable-step integration method is used. (Fixed steps are usually faster anyway when simulating systems subjected to random noise.)

With discrete simulation, on the other hand, different runs of the model tend to follow vastly different paths. Even the entities in existence at any given instant are sometimes quite different from run to run. Thus one must be doubtful about the possibilities of replication in that field. Even there, though, there would probably be cases where it could be made to work.

Table 1 summarises this writer's view of how replication increases the broad classes of application for vector computers.

Since there has been little sign of replication being applied to simulation it was decided to provide a demonstration on a problem of that type.

Table 1. Problems suitable for vector machines

1.	Systematically structured
1.1	Partial differential equations
	Especially in three space dimensions and time.
	Problem may be stated in some other, more user-oriented form.
1.2	Large matrix problems
	Often derived from partial d.e. problems.
2.	Replicated problems
2.1	Signal processing
	Replicated fast Fourier transforms and the like.
2.2	Modelling dynamic systems
	With noise and/or optimisation.
	Continuous and some discrete simulation.
2.3	Optimisation
	Several varieties to consider.

3. A SIMULATION EXAMPLE

3.1 The test problem and its results

The test problem was a continuous simulation of a servomechanism (position control system). The part whose position was being controlled was subject to randomly varying disturbing forces, requiring the servo's accuracy to be evaluated by averaging the results of a large number of simulation runs. The equations of the model were straightforward, apart from some rather complex non-linear functions which were needed to represent saturation and Coulomb friction.

First the problem was programmed, debugged and timed on a more conventional type of machine, a Cyber 76. The program structure used is standard for this class of problem, providing the flexibility needed in simulation while being efficient on scalar computers.

On the Cyber 76 the speed achieved was about 2.4 Mflops – millions of floating point operations per second. The figures quoted here are a little arbitrary since they depend on what one counts as a floating point operation. Is taking the absolute value a full-fledged floating point operation? And how many operations do functions like SIN and EXP count as? Operation counts were weighted roughly according to the relative times taken on the Cyber 76, but the results are probably meaningful mainly as measures of relative rather than absolute speeds.

Tests on the Cyber 205 began with a program differing as little as possible from the Cyber 76 version. The program was then progressively modified to exploit fully the machine's capabilities. Initially the speed was only 2.1 Mflops (slower than the Cyber 76!) but by the end it was up to 110 Mflops (46 times faster than the Cyber 76). This is an extreme case of what many have found when converting to this type of architecture – existing programs written without thought for vectorisation may yield disappointing speeds at first, but after some effort the results can be spectacular.

There were three main stages of improvement: vectorising the main DO loop, vectorising the non-linear functions, and vectorising random-number generation.

3.2 Vectorising the main DO loop

The first step was to try the compiler's automatic vectorisation option. It gave absolutely no improvement. This was expected – the reason will be clear from the following code. (This is not the servo problem but a hypothetical problem that is easier to explain.)

(1) Routine calling 1000 times for a run of the model:

```
      :
    DO 1 I = 1, 1000
  1 CALL MODEL (COST (I))
      :
```

```
(2) Subroutine for one run:
        SUBROUTINE MODEL (COST)
           :
        R = RANF()
           :
         X = U*V+R
           :
        COST=----
        RETURN
        END
```

Since the FORTRAN compiler looks at only one subroutine at a time, it misses the fact that the equations of the model are embedded in a DO loop and so sees nothing to vectorise. A necessary preliminary must be to move the DO loop inside the subroutine as follows.

```
(1) Routine calling once for 1000 runs of the model:
           :
        CALL MODEL (COST)
           :
(2) Subroutine for 1000 runs:
        SUBROUTINE MODEL (COST)
        REAL COST (1000), R(1000)
       +  U(1000), V(1000), X(1000)
       +  ...
        DO 1 I=1, 1000
           :
        R(I)=RANF()
           :
        X(I) = U(I)*V(I)+R(I)
           :
      1  COST(I)=---
        RETURN
        END
```

The need for the above type of change illustrates the general point that the compiler is only able to peform optimisation and vectorisation on a local scale. Changes of a more global nature, which are often the most effective, are up to the programmer.

Where the model is not all contained in one subroutine, but is distributed throughout a hierarchy of subroutines, still further demotion of DO loops is necessary, along these lines:

```
   Before:                    After:
   SUBROUTINE MODEL           SUBROUTINE MODEL
   DO 1 I=1, 1000             DO 1 I=1, 1000
   ---                        ---
   ---                        ---
   ---                      1 ---
   CALL SUBMOD                CALL SUBMOD
                              DO 3 I=1, 1000
   ---                        ---
   ---                        ---
 1 ---                      3 ---
   RETURN                     RETURN
   END                        END
   SUBROUTINE SUBMOD          SUBROUTINE SUBMOD
                              DO 2 I=1, 1000
   ---                        ---
   ---                        ---
   ---                      2 ---
   RETURN                     RETURN
   END                        END
```

Further breaking up of loops may then be necessary where there are IF tests, but this will be discussed in the next Section.

Once the compiler is able to see some DO loops, it is able to vectorise them automatically if they satisfy certain rules. Alternatively, one can vectorise them by hand using the special vector syntax provided in the Cyber 205 dialect of FORTRAN. This second method was chosen, mainly as a means of learning more about what can and cannot be done by the machine's vector instructions – so as in future to be better able to write code that would vectorise automatically.

With the experience gained it is believed this problem could now be handled by the automatic vectorisation method to give almost as good results, with the advantage of much less use of non-standard FORTRAN.

As was to be expected, this primary stage of vectorisation made a big difference and gave an order-of-magnitude improvement in speed.

3.3 Vectorising the non-linear functions

The non-linear functions typically had the following pattern:

```
 DO 1 I=1, 1000
        Equations computing some logical condition C as a
        function of X(I).
 IF (C) THEN
     Equations computing result F(I) as one function of X(I).
 ELSE
     Equations computing result F(I) as some other function of
     X(I).
 END IF


1 CONTINUE
```

If C were a fixed condition independent of X(I), one could take its computation outside the loop, split the remainder into two shorter loops and use the IF test to control which of the two to execute. This would give loops requiring nothing but uninterrupted operations on elements in consecutive memory locations – just what vectorises most easily. When, as here, C varies with each iteration of the loop, the simplest method proceeds along these lines:

```
Compute logical bit vector C as a function of vector X.
Compute vector F1 as the first function of vector X.
Compute vector F2 as the second function of vector X.
Call library function Q8VMASK to mask elements of either F1
      or F2 across to the result vector F according to the bit
      pattern of C.
```

The computations of C, F1 and F2 can be programmed using either explicit vector syntax or automatically vectorisable DO loops. The masking requires non-ANSI syntax using one of the vector functions available in the FORTRAN library.

The above method has the disadvantage that each component of the result vector has to be computed by both sets of equations and the value not needed thrown away. This may not matter if the value most often thrown away is much cheaper to compute than the other. In the servo problem it was the other way round, so it became worthwhile changing to the following method:

```
Compute logical bit vector C as a function of vector X.
Call Q8SCNT to count number of 1s in vector C to allow the
      correct lengths to be specified for vectors X1, X2, F1 and
      F2 below.
Call Q8VCMPRS to compress those elements of X for which
```

the corresponding element of C is a 1 into contiguous locations in vector X1.
Call Q8VCMPRS to compress the remaining elements of X into vector X2.
Compute vector F1 as the first function of vector X1.
Compute vector F2 as the second function of vector X2.
Call Q8VMERG to merge elements from either F1 or F2 into result vector F according to the bit pattern of C.

This method is somewhat more complex than the first and requires even more use of non-standard FORTRAN, but it can be significantly faster even after allowing for the overhead of compressing.

As can now be seen, where alternative paths in the algorithm being replicated are simple and occur on only a local scale within one routine, as was the case with these servo functions, the problem of keeping the different replicates in step is dealt with simply in the course of vectorising. It is only where the alternative paths are on a much broader scale, with branches within branches, that it becomes difficult to arrange for all replicates to be doing the same thing at the same time.

3.4 Vectorising random-number generation

Initially, the library subroutine VRANF was used for generating vectors of random numbers. However, the program did not run as fast as expected and it turned out that 70% of the time was being spent generating random numbers. (This compared with only 4% on the Cyber 76.)

VRANF is so slow because although it produces a vector of random numbers it generates them using a scalar seed. To generate them by vector methods, it is essential to have a vector of seeds. If vector computers are to be exploited on stochastic problems it is important to have available a fully vectorised random-number generator.

Developing and testing such a generator is a sizeable task. For the time being, a makeshift generator was devised. It uses a replicated set of generators using the VRANF algorithm working in parallel on a vector of seeds. The initial numbers in the seed vector have to be set randomly and independently by some other generator. One was designed using the rules given in Volume II of D. E. Knuth's *The Art of Computer Programming*.

4. CONCLUDING REMARKS

There is much more to do to explore the possibilities and limitations of replication. It was easy to apply in the servo example, but that problem was chosen for its suitability.

A systematic set of rules needs to be developed for replicating the various basic types of programming structure. An ultimate aim could be a pre-processor to do automatically most of the work of replicating any given algorithm.

Research into replication and other forms of vectorisation is important, not only because it will help get the most out of the present generation of vector processors, but for two other reasons.

(1) Future computers generally are likely to make more use of vector and similar forms of parallelism. This seems likely to extend even down to personal computers.

(2) Vectors and matrices are the natural data objects to use in many applications, and higher-level languages (including future versions of standard FORTRAN) will include them. As others have said, we must all learn to 'think vectors'.

Acknowledgement

I wish to thank Dr Peter R. Taylor for drawing my attention to some of the existing work on replication.

REFERENCES

1. B. J. Alder and D. M. Ceperley, Some programming aspects of quantum Monte Carlo calculations. *Proceedings of Symposium on Cyber 205 Applications, Colorado, August 1982.* (Available from Conferences and Institutes, Colorado State University, Fort Collins, Colorado 80523, USA.)
2. M. F. Guest and S. Wilson, (eds), Electron correlation. Daresbury Study Weekend, 17–18 Nov. 1979. (Proceedings available from SERC Daresbury Laboratory, Daresbury, Warrington WA4 4AD, UK.)
3. M. F. Guest and S. Wilson, The use of vector processing in quantum chemistry – experience in the UK. In *Supercomputers in Chemistry*, edited P. Lykos and I. Shavitt. American Chemical Society, Washington. (1981).
4. D. Hegarty and G. van der Velde, A quantum chemical program system. *Proceedings of Symposium on Cyber 205 Applications* (see Ref. 1).
5. D. Hegarty and G. van der Velde, Integral evaluation algorithms and their implementation. *International Journal of Quantum Chemistry* **23**, 1135–1153 (1983).
6. V. R. Saunders and M. F. Guest, Applications of the CRAY-1 for quantum-chemistry calculations. *Computer Physics Communications* **26**, 389–395 (1982).
7. P. R. Taylor, *Supercomputers in Quantum Chemistry – A Survey*. Technical Report 82-TR01, CSIRO Division of Chemical Physics, PO Box 160, Clayton, Victoria 3168, Australia.
8. C. Temperton, Fast methods on parallel and vector machines. *Computer Physics Communications* **26**, 331–334 (1982).