A TECHNIQUE FOR EXPLOITING PARALLELISM IN METEOROLOGICAL SOFTWARE

D.J. Catton
Strand Software Technologies Ltd
Markyate, UK.

Summary: A few problems can be solved on a multi-processor computer by running simultaneously a collection of quite independent sub-programs, or processes. In most problems, however, the parallel sub-programs need to communicate intermediate results between themselves. This communication makes programming more complex because data may be read by a recipient more than once, or alternatively, missed altogether. Strand is a new concurrent programming language for MIMD type machines which has been designed to relieve programmers from the burdens of co-ordinating communication between many processes running simultaneously. A Strand program can also be written as a *harness*, to co-ordinate communication between processes written in Fortran or 'C'.

This paper summarises the important features of Strand and describes in outline an experiment in which a Strand harness was written to adapt a weather forecasting program to parallel running.

1. INTRODUCTION

Multi-processor MIMD machines where each processor can run a different program offer great flexibility in speeding-up computations through parallel operation. A class of problems, such as those based on Monte Carlo simulations, can be solved on MIMD machines by using a collection of independent sub-programs running in parallel. In most cases however, a parallel solution has to be achieved by having a collection of sub-programs which communicate intermediate results amongst themselves as the computation proceeds. Data interchange is usually needed in matrix manipulation and in the numerical solution of partial differential equations, see for example, Fox *et al* (1988).

Data interchange has to be controlled otherwise some data passed on might be read

more than once, or alternatively, it might be missed altogether. In parallel programs based directly on sequential languages such as Fortran or 'C', it is usually necessary for the programmer to take care to arrange the orderly transfer of data. This can be a burden and can lead to obscure programming errors. The Strand concurrent programming language, Foster and Taylor (1990), has been designed to take over the burden of arranging orderly data transfer between processes.

Strand can also control the running and intercommunication of routines written in Fortran or 'C' and it is possible to write a harness program in Strand which controls the running of a collection of process routines written in C and Fortran. In this way the numerical parts of a program can be written in well established formats whilst the supervisory parts needed for parallel computation draw on the new features provided by Strand. This approach to constructing a parallel program is sometimes called *bilingual programming*. The code needed to mediate between the data storage formats needed by Strand and those needed by Fortran is supplied as a library routine with each copy of Strand.

2. FEATURES OF STRAND

A Strand program is based on a collection of process definitions which call upon each other. Program execution is started by a call to one, or more, processes. An example of a process definition is -

```
twice(Number, Result) :- Number > 0 |
                                Result is 2*Number.
twice(Number, Result) :- Number =< 0 |
                                Result := 0.
```

This definition has two cases which are selected for execution by the *guards*, Number > 0 | and Number =< 0 |. Notice that the computed result has to be assigned to a named variable and cannot be left anonymous. A program could be started by the call, twice(2,Answer) when the first case would execute to leave the result 4 in Answer. There can be several operations on the right-hand side, Strand seeks to execute all of these in parallel if possible.

Another example of a call is, twice(2,W),twice(W,Answer). This call would nominally start two instances of the twice process executing in parallel. However, Strand schedules its computations using dataflow rules and the second instance of twice process would not be started until the value of W had first been computed by

the first process. Figure 1 illustrates the call as a network of two processes communicating through the shared variable W.

2.1 Stream Data

The above call had to transfer only one item of data between the two processes. In many problems more than one item has to be transferred, for example, values might have to be transferred on every iteration. To handle the transfer of a succession of items Strand uses a *stream* construct.

The stream construct depends on placing data into list structures and manipulating these structures with a standard set of conventions. An example of a list used in the stream construct might be, [1,2|T]. This denotes the list starting with the numbers 1 and 2 and continuing with further numbers not yet computed. The list of further number will eventually appear in the variable T.

One case of a program to double each number in a stream might be,

```
streamTwice([H|T],ListResult) :- H > 0 |
                                 Temp is 2*H,
                                 ListResult := [Temp|ListResult1],
                                 streamTwice(T,ListResult1).
```

The stream program deals with the first item on the stream and then spawns a new process to manipulate the further items on the stream. This form of program gives the interesting effect that in the call, streamTwice([1,2,3],ListResult), the starting call generates a process to manipulate 1 and at the same time spawns a process to deal with the list, [2,3]. The second process similarly manipulates 2 and spawns a process to to deal with [3]. The effect is to vectorise the computation automatically, see figure 2.

More than one process can receive the same stream input and diverse networks of processes can be set-up.

3. THE MM4 WEATHER MODELLING PROGRAM

MM4 is the code of a medium scale weather modelling Fortran program developed at Pennsylvania State University and the U.S. National Center for Atmospheric Research. MM4 computes the time evolution of a set of partial differential equations in the usual way by dividing the atmosphere into a 3-dimensional grid of points.

Some experiments have been done by I. Foster, D. Joerg and R. Stevens of Argonne National Laboratory, Illinois, U.S.A. to adapt the MM4 code so that it could run on a multi-processor machine. This work will be reported in detail elsewhere.

In the parallel computing experiments the basic data had to be divided and spread over the available processors. In effect, the atmosphere over an area roughly the size of the continental United States was divided into 16 horizontal layers. Each layer contained 61x46 grid points, each grid area corresponding to an 80x80 km square. The values associated with a grid point at time $t+1$ could be computed in the usual way from the values associated with several neighbouring grid points at time t. The atmosphere grid was decomposed along two dimensions into sub-grid layers of air.

Calculating the new, time $t+1$, values associated with the points of a sub-grid was done by a one process for each sub-grid. Of course, several computational processes had to be allocated to each available physical processor because the available machines had 64 or fewer, processors. The new values associated with points well within the sub-grids could be found with the data available at each processor. On the other hand, to update points near the sub-grid boundaries, values from neighbouring sub-grids would be needed. These values would be obtained either from processes on the same processor or from processes on other processors.

4. PROGRAM DESIGN

The essential part of the adapted MM4 program was the Fortran code which computed the updated values of the points of a single sub-grid. The rest of the program was concerned with initialising the sub-grid values and with running each iteration step. At each iteration step the boundary values of each sub-grid had to be communicated to designated neighbours and in turn, appropriately received.

The sub-grid code was largely that of the MM4 Fortran. It had to be adapted to eliminate the use of common storage since each parallel process runs in a self-contained way. Strand and Fortran store data in different formats so a format conversion is carried out when Strand passes data to the Fortran routine and when data is returned.

A simplistic approach to writing the harness would be to write a call which would

immediately set-up the required sub-grid processes. This call would have already divided up the initial data for the sub-grids. Writing this call would be heavy work and it would have to be written differently for each size of machine. A better approach is to delegate the setting-up work to a program.

A standard design for setting-up programs is the based on the manager/worker approach. In this arrangement a manager process is started first, this reads in details of the number of available processors and allocates a worker process to each processor. The manager then reads in details of the parallel processes which comprise the program. The manager then starts running the program by allocating program processes to be run by each worker processes. When a worker finishes it passes back results to the manager and asks for another process to run.

In the MM4 adaptation the main processes are those updating the sub-grid values, but there are also processes which decompose the initial data. The manager receives not only values but also error estimate information. A diagram illustrating a possible general arrangement of a manager/worker network is shown in figure 3.

5. CONCLUSIONS

The experimental results to be published by Foster, Joerg and Stevens will demonstrate that a worthwhile speed-up of the MM4 computations were obtained on all the different multi-processor machines which were available. This shows that use of a manager/worker program design with the controlling portion written in Strand and the numerical routines written in Fortran is both portable across different machines and also efficient in realising the benefits of parallel computation.

A major advantage of the kind of bilingual program framework discussed above is that it is easy to change the basis of the computation, for example the grid size can be reduced in parts of the atmosphere where change is greatest. The changes can be made adaptively as the computation proceeds because Strand can create new worker processes dynamically at any time during computation and can invoke the Fortran processes with new grid size parameters.

REFERENCES

Fox, G., M. Johnson, S. Otto, J. Salmon, and D. Walker, 1988: *Solving problems on concurrent processors, Volume 1*,Englewood Cliffs, New Jersey, Prentice-Hall, 592pp.

Foster, I. and S. Taylor, 1990: *Strand, new concepts in parallel programming*, New Jersey, Prentice-Hall, 323pp.

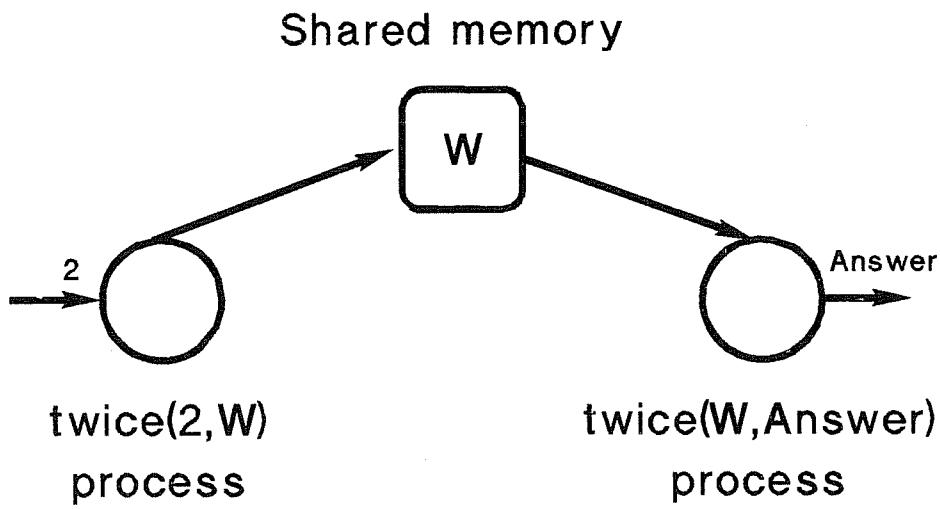

Fig 1. Process network of call twice(2,W),twice(2,Answer)

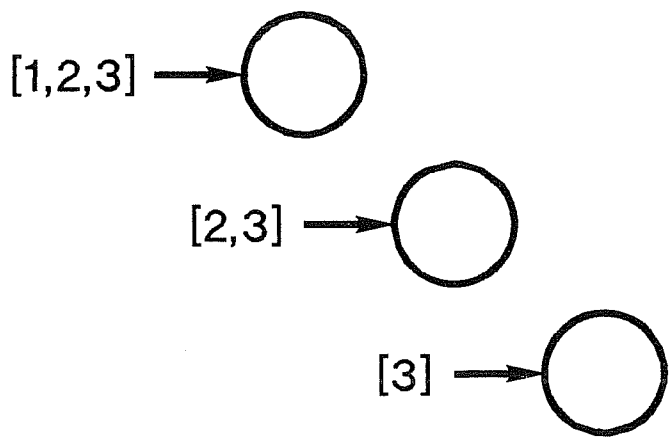

Fig 2. Vectorising effect of stream processing