COMPUTER SCIENCE TECHNICAL REPORT

No. 92-14

Increasing the Efficiency of Vectorization Through the Use of Multiple Sub-Zones with Automatically Mutually Exclusive Nodes

A Case Study Through the Ising Problem

Evan B Golub
David M Arnow

BROOKLYN COLLEGE

Department of Computer and Information Science
Brooklyn College of C.U.N.Y.
Bedford Ave. and Ave. H
Brooklyn, New York 11210

1) Introduction

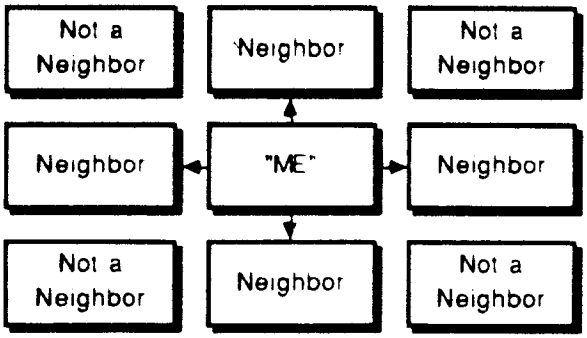
In parallelizing algorithms, care must be taken to guarantee that no two processors will be updating a given variable at the same time. One technique which is used, is doing work in secondary structures, and then combining partial results at a later time. This adds the step of recombining the data. In this paper we look at how the same ends can be accomplished without the use of secondary workspace, but rather by logically restructuring the data to be worked on. The results obtained show a decrease in the sequential code's run time, which in turn leads to a decrease in the parallel code's run time.

The specific example used to demonstrate and compare these two techniques is the Ising Problem. The empirical results were obtained by running the code on an Alliant FX/8 with eight processing elements, each having thirty-two vector processors, all accessing the same shared memory.

2) Presentation of the Basic Ising Problem

The "Ising" problem has been shown to be a good benchmark for testing techniques in parallel programming. The problem consists of a multidimensional matrix in which each cell can either lose or gain energy, to or from a neighboring cell according to a specified set of rules. For this case, the matrix is two dimensional, with the size of 128 x 64. The rules regarding the cells are as follows. The corner cells have no energy value, and have no ability to gain or lose energy. The wall cells have an unspecified, "infinite" amount of energy, and can "lose" one energy unit per time unit to a neighboring cell. The interior cells begin with no energy, but can gain energy from its neighbors. After it has energy of its own, it can then give off its energy to any one of its neighbors.

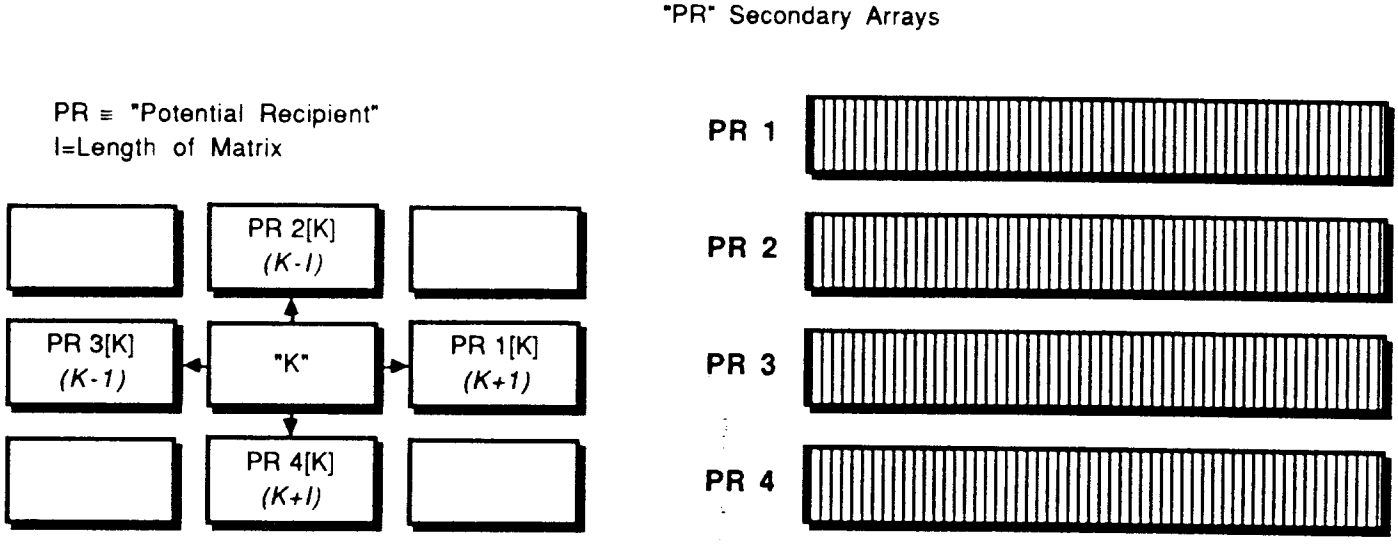
A neighboring cell is defined as a cell which is adjacent, either horizontally or vertically. Cells which are diagonally adjacent are not considered to be neighboring cells.



At each "tick" of the clock, each cell performs its own energy transfer instantaneously. This allows us to decide the energy transfers "before" the time of transfer, and prevents dependencies between cells during the passing of a single time unit. (During one time cycle, the entire matrix is updated according to the energy transfers to and from each active cell.) The problem which does arise is that a single interior (non-border) cell can be updated by up to four different neighboring cells during one cycle. In order to have accurate results, it has to be guaranteed that parallelizing the code will not lead to multiple actions on a single cell at the same time. A straight forward method of accomplishing this would be to have a mutual exclusion lock in place for each cell of the matrix. This however leads to enormous overhead, the potential of severely serializing the execution of the program, and eliminating the possibility of vectorization. The two techniques which follow deal with the problem of working with these interior cells in two very different manners. The first technique creates secondary data structures to hold intermediate updates, and then merges them to the matrix. The second technique logically restructures the matrix to guarantee that no two updates can effect a given cell while being done in parallel.

3) Description of the First Method : Secondary Structures

In the first method which will be discussed, changes are not made directly to the cells of the matrix. Instead, a second set of structures is created which will hold four update values for each cell. These four cells represent how much energy a given cell will give of to its neighbors (up, down, left and right). As each cell is "asked" how much energy it will release, and which neighbor it will release it to, that information is saved in the appropriate update location in the secondary structure and that amount of energy is removed from the cell being "polled.". Then, after the entire matrix has been "polled", each cell's new value is computed based on the original values and the secondary structures' information.

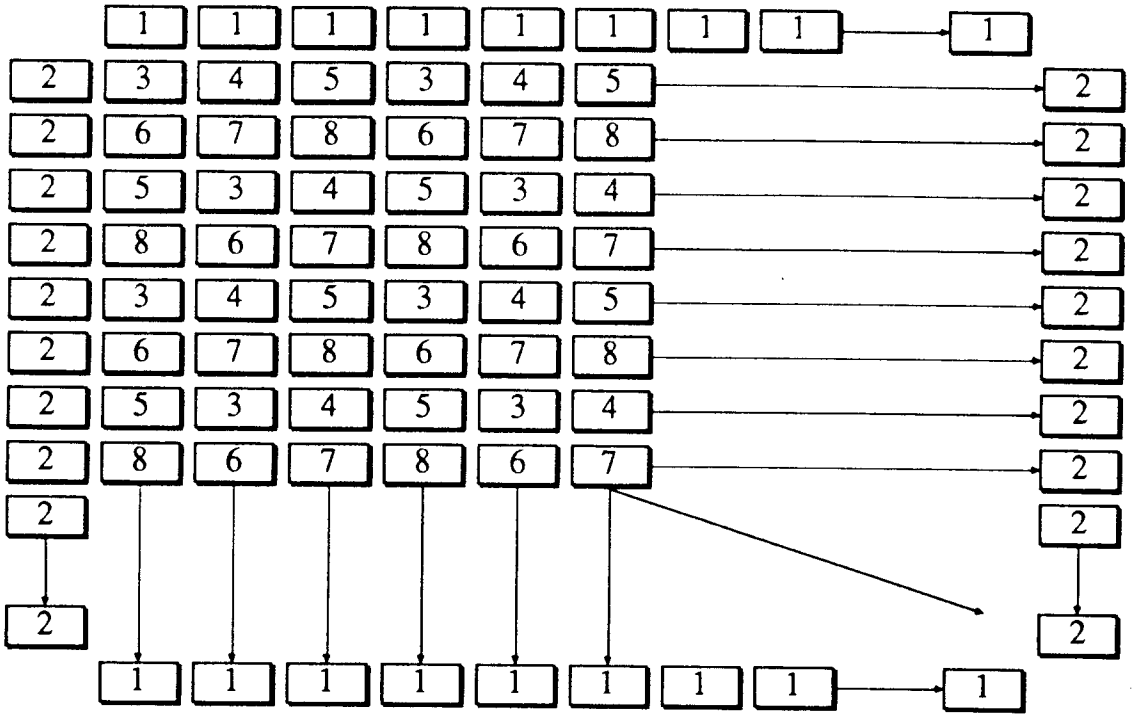


Recalculation : matrix(i) = matrix(i) + PR1(i+1) + PR2(i-I) + PR3(i-1) + PR4(i+I)

A single line of code does this calculation for each cell, so there is no risk of one cell being updated by multiple instructions at the same time when the code is parallelized. This method is quite effective, however, there is an added loop where the cells' new values are calculated based upon the information in the secondary structures. This is where the motivation for the second method came from.

4) Description of the Second Method : Logical Restructuring

The motivation for the second method which will be discussed was to attempt to eliminate the need of an additional loop to merge all of the update information and recalculate the matrix values. The solution was to have each cell updated as the energy transfer decisions were made. The problem with this arises when the code is parallelized, and a given cell could be in the process of being updated by up to four neighboring cells. To overcome this problem, the matrix was logically restructured into eight zones. Two of these zones equally divide the wall cells, and the remaining six zones equally divide the interior cells. These zones are organized in a way in which no cell in a given zone can be effected by energy transfers made by any other cell in that zone.



Each of the six zones need to be computed serially, (eg:Zone2, ..., Zone7), but the cells within each zone can be computed in parallel. This eliminates the need for the secondary structure for updates, and therefore the need for the extra loop to recombine the information on updates. It can appear that serializing the

six zones would serialize the code, but the only loss is the startup cost at the beginning of each of the six zones. If the zones are large enough, which they are in this case, then the effect is negligible.

5) Empirical Results

In order to fully compare these two techniques it was only necessary to time the code which dealt with energy transfers of interior cells, where one version would simulate energy transfers using Method #1 and the other using Method #2. Both versions were run with the vector-concurrent flags turned off and then on. In order to get accurate timing data, several procedures were followed in the runs. Primarily, the only part of the code to be timed was the altered procedure. However, since real time was being tested, and this can be seriously effected by system utilization and other factors, all four versions of the program were run simultaneously so that they would all be run under the same set of circumstances. Additionally, 20 repeated trials were run and averaged together in order to get a fair representation.

The following is the accumulated data.

Method#1 : Sequential	78.0957	seconds
Method #2 : Sequential	74.0009	seconds
Method #1 : Parallel	7.4779	seconds
Method #2 : Parallel	6.8599	seconds

Speed-up for Method #1 from Sequential to Parallel :	10.4435
Speed-up for Method #2 from Sequential to Parallel :	10.7875
Speed-up for Sequential runs of Method #2 over Method #1 :	1.0552
Speed-up for Parallel runs of Method #2 over Method #1 :	1.0901

Of the above numbers, the most relevant one to the original premise for creating the second method is the speed-up of the parallelized run of Method #2 over Method #1. The speed-up is approximately 9%.

6) Conclusions

As shown by the empirical results, the primary difference between these two methods is that the sequential code is reduced by eliminating the need for a loop to merge update data. While in this case the relative amount of time spent in this loop is small, it is possible to imagine situations where the time utilized for merging secondary structures' data into the primary matrix is a substantial part of the work done. In these situations, creating a logical restructuring by observing the traits of the matrix to be operated upon could lead to significant speed-up of the serial code, and in turn, of the parallel code. The use of visual representations is helpful in this process, but it could be possible to automate the procedure by writing a program to evaluate and find the proper zones for a given situation.

Appendix A-1 : No Sub-Zones In Use (Secondary Storage Matrix Used Instead)

```
      SUBROUTINE SIMULATE
      IJLIM = 128*64
      NRAN=5


      DO 20 IJ=1,IJLIM
         NEW(IJ,1) = 0
         NEW(IJ,2) = 0
         NEW(IJ,3) = 0
         NEW(IJ,4) = 0
   20 CONTINUE

      NRAN=NRAN-1

CVD$L NODEPCHK
      DO 70 IJ=1,IJLIM
         N = 0.5 + RNS(NRAN+IJ)*IGR(IJ,ICUR)
         IGR(IJ,NEXT) = IGR(IJ,NEXT) - N
         M = 1.0 + (4.0-1.0e-10)*RNS(NRAN+IJ+IJLIM)
         NEW(IJ,M) = N
   70 CONTINUE

      NRAN=NRAN+IJLIM*2+1

      DO 90 IJ=1,IJLIM
         N1 = 0
         N2 = 0
         N3 = 0
         N4 = 0
         IF (IJ .LT. IJLIM)     N1 = NEW(IJ+1,1)
         IF (IJ .GT. 64)        N2 = NEW(IJ-64,2)
         IF (IJ .GT. 1)         N3 = NEW(IJ-1,3)
         IF (IJ .LT. IJLIM-64) N4 = NEW(IJ+64,4)
         IGR(IJ,NEXT) = IGR(IJ,NEXT)+N1+N2+N3+N4
   90 CONTINUE


      END
```

Appendix A-2 : Sub-Zones In Use

```
      SUBROUTINE SIMULATE
      IJLIM = 128*64
      NRAN=5


CVD$L NOCONCUR
CVD$R NOSYNC
      DO 5270 IOUTER=1,6
CVD$L CONCUR
      DO 5279 IPOS=1,1302
         I=ISLOT(IOUTER,1,IPOS)
         J=ISLOT(IOUTER,2,IPOS)
         IJ=(J-1)*64 + I
         N=0.5+RNS(NRAN+IPOS+1302*(IOUTER-1))
     +                        *IGR(IJ,ICUR)
         M=1.0+(4.0-1.0e-10)*RNS(NRAN+7812+IPOS+1302
     +                         *(IOUTER-1))
         IGR(IJ,NEXT)=IGR(IJ,NEXT)-N
         II = I + IOFFSET(M)
         JJ = J + JOFFSET(M)
         IIJJ=IJ+IOFFSET(M)+64*JOFFSET(M)
         IF (.NOT.(II.LT.2.OR.II.GT.63.OR.JJ.LT.2.OR.JJ.GT.127))
     +    IGR(IIJJ,NEXT)=IGR(IIJJ,NEXT)+N
 5279 CONTINUE
 5270 CONTINUE


      END
```