Stanford Exploration Project, Report 65, January 13, 1998, pages 253–262

Stanford Exploration Project, Report 65, January 13, 1998, pages 253–262

Velocity Analysis on the Connection Machine

Guido Kneib[1]

keywords: *algorithm, velocity, computing, arrays*

ABSTRACT

Implemented in slightly different ways on one of the new massively parallel computers, the Connection Machine, the performance of Jon Claerbout's pixel-precise velocity analysis algorithm can differ by much more than one order. The performance depends strongly on the way the Fortran programmer maps the data on the machine.

INTRODUCTION

There are good reasons to believe that parallel computers will play a major role in future geophysical computing. First, the traditional serial computing is already close to its physical limits (Boghosian, 1990) and second, physical models and architectures of computers seem to develop towards similar topologies (Hillis, 1985).

Boghosian gives a good general introduction to one of the fastest, highly parallelized computers, the Connection Machine (CM). Most of the literature dealing with the Connection Machine describes special applications in geophysics, like those of Mora (1989) or Delany (1988) or in other fields (Long, 1989). Myers and Adams (1989) investigate the CM's performance in more detail.

The authors usually discuss only high-performance implementations. From the didactical point of view, comparing inefficient programs with faster versions can be more instructive. In this paper I will show the importance of an intelligent mapping of the data to the virtual processors, to reduce interprocessor communication and to achieve a high degree of parallelism.

I will demonstrate this by means of Claerbout's pixel-precise velocity analysis algorithm (Claerbout, 1989, and Claerbout, 1990) which will prove as a difficult choice.

[1]**email:** not available

THE CONNECTION MACHINE

The Thinking Machines Corporation, Cambridge, Massachusetts, released the first CM-2 in April 1987. Since then, more than 40 units have been installed in the US and Europe. The CM-2 is one of only few parallel computers using the one-data—one-processor concept and concentrating on increasing the number of processors rather than using faster processors.

The most important feature of the CM's architecture is the concept of virtual processors (VPs). Each element of a parallel variable (usually an array element or an array dimension) is associated with one virtual processor. VPs are analogous to virtual memory: In either case the user can address a nearly infinite number of processors, or memory cells, independent of the actual hardware configuration.

If as usual the number of physical processors N_P is bigger than the number of virtual processors N_V, then every VP is related to one physical processor. If $N_V > N_P$, at least some physical processors have more than one VP associated with them. The ratio $\frac{N_V}{N_P}$ is called the VP-ratio. Because calculations can involve only the currently active set of VPs, i.e. the VPs currently present in physical processors, one has to switch from one VP-set to the next, if the VP-ratio is bigger than one. This is called VP-looping. Extensive VP-looping is costly, because it requires a physical shift from data between the external storage system and the processors' memories.

The high-level language-user can always think of the CM processors as representing a Cartesian grid, because Cartesian coordinates can be embeded in the 12-dimensional hypercube that forms the CM's hardware interconnectivity. The n-dimensional grid of VPs is described by the processor's geometry and can be defined by the user according to the problem being addressed. You will find an example in my first program sample.

The user accesses the CM via a front-end computer. All editing, compiling, and linking are done on the front end. Programs run on the front end, too, but parallelizable program parts are executed on the CM.

The Connection Machine itself consists of one, two, or four portions a 8192 or 16384 processors. Each of these parts is accessable to different users because it has its own sequencer that interprets the parallel instructions sent by the front end and activates its processor portion accordingly. I always used 8k processors, the smallest CM-portion available. Each CM processor has its own 8k memory and arithmetic logical unit.

Programming on the CM

CM-users have the choice between three programming languages: C^*, Fortran and $^*LISP$. I used only CM-Fortran. CM-Fortran is an extension of Fortran77. It includes the array facilities of Fortran8x but has some restrictions where the storage order would be crucial to the program. CM Fortran features are, among others, intrinsic functions, and arithmetic and logical operations involving whole arrays or array portions.

The code generated by the CM Fortran compiler generates code that directly calls the parallel instruction set (Paris) and therefore allows computations to be made on the CM. Explicit use of Paris allows a direct control over how parallel operations are done. But to produce more efficient code than the Fortran compiler does automatically, the programmer needs some experience.

Important restrictions for a parallelization are

- Statements that affect the flow of execution within a program must translate into code that is executed entirely on the front-end machine.
- Where an instruction is executed depends on where its operands have been allocated memory space. Scalars are always allocated on the front end and arrays can be allocated either on the CM or on the front end, or on both.

Arrays can be stored on the CM explicitly, when compiler directives are used, or implicitly, if an array is used in an explicit Paris call, an intrinsic CM array function, or a Fortran8x array command. Examples of compiler directives can be found in the program on the next page.

COMPARISON OF EXAMPLE PROGRAMS

FD-schemes or simple matrix multiplications run very fast on the CM. I chose Jon Claerbout's new pixel-precise velocity analysis program (Claerbout, 1990) for an implementation. The program "Vspray" has the following loop structure:

```
do iz = 1, nt                                   ! loop over tau
   do it = iz, nt                               ! loop over time
      do ix = 1, nx                             ! loop over offset
         s = slowness(tau,t,x)
         do s = smin, smax                      ! loop over slowness
            zs(tau,s)=zs(tau,s)+data(t,x)       ! smear data
end do
```

Besides calculating the RMS velocity from a seismogram, "Vspray" performs a smearing along the velocity axis. The use of four combined do-loops should make parallelization very effective, but the access of different array elements with four different indices in the innermost loop should lead to extensive interprocessor communication. In my opinion, these features make "Vspray" an interesting choice for a test. "Vspray" is relatively costly on serial computers, because the computing time is roughly proportional to ${N_t}^2 N_x/2$.

The following shows my first trial, "Vspray1". The algorithm is unchanged but is simply translated to CM Fortran; arrays are used for variables needed in the innermost loops. Via compiler directive all arrays are allocated on the CM and are completely parallel, using the so-called NEWS geometry that places subsequent array elements in a nearest neighbor fashion together. Only the offset dimension is serial. This exception means that each time slice is present only in one virtual processor. Making the offsets parallel increases the amount of interprocessor communication and slows down the program. Some table variables are aligned with the input $tx()$, to save communication time. Aligned arrays have the same "home"; i.e. they are assigned to the same virtual processor.

```
c
c Program Vspray1
c
      subroutine vspray( nt,dt,t0, nx,dx,x0, tx, ns,ds,s0, zs)
      integer it, nt, iz, nz, ix, nx, is, ns, isp, ism
      integer isma(nt,nx), ispa(nt,nx)
      real z, dz, z0, t, dt, t0, x, dx, x0, s, ds, s0, sm, sp, dthalf
      real xx(nt,nx), xsq(nt,nx), zsq(nt,nx), tsqp(nt,nx), tsqm(nt,nx)
      real con(nt,nx), sca(nt,nx), sma(nt,nx), spa(nt,nx)
      real tx(nt,nx), zs(nt,nx), tt(nt, nx), scale
c
c all CM Fortran compiler directives begin with:  CMF$
c
c
c define the layout, i.e. the geometry of the virtual
c processors for the input and output array
c :NEWS         is a parallel nearest neighbor geometry
c :SERIAL       makes the dimension serial
c
CMF$  LAYOUT tx(:NEWS,:SERIAL), zs(:NEWS,:NEWS)
c
c align certain table arrays to the input array to reduce the communication
c
CMF$  ALIGN tt(it,ix) WITH tx(it,ix)            ! time array
CMF$  ALIGN tsqm(it,ix) WITH tx(it,ix)          ! (t-dt/2)**2
CMF$  ALIGN tsqp(it,ix) WITH tx(it,ix)          ! (t+dt/2)**2
CMF$  ALIGN zsq(it,ix) WITH tx(it,ix)           ! tau**2
CMF$  ALIGN xx(it,ix) WITH tx(it,ix)            ! offset
CMF$  ALIGN xsq(it,ix) WITH tx(it,ix)           ! offset**2
CMF$  ALIGN con(it,ix) WITH tx(it,ix)           ! scaled seismogram
CMF$  ALIGN sca(it,ix) WITH tx(it,ix)           ! scale factor
CMF$  ALIGN spa(it,ix) WITH tx(it,ix)           ! upper slowness limit
CMF$  ALIGN sma(it,ix) WITH tx(it,ix)           ! lower slowness limit
CMF$  ALIGN ispa(it,ix) WITH tx(it,ix)          ! upper slowness indices
CMF$  ALIGN isma(it,ix) WITH tx(it,ix)          ! lower slowness indices
```

```
c
c initialization of scalars and arrays
c
      nz=nt
      dz=dt
      z0=t0
      dthalf=dt/2.
      zs = 0.                           ! initialize the whole output array
      do 23002 it= 1,nt
             t = t0 + dt*(it-1)
             tt(it,:) = t                       ! ':' means: do the same for
             tsqp(it,:) = (t+dthalf)**2         ! the complete dimension
             tsqm(it,:) = (t-dthalf)**2
23002        continue
      do 23004 ix= 1,nx
             xx(:,ix) = x0 + dx*(ix-1)
23004        continue
             xsq = xx*xx
      do 23006 iz= 1,nz
             zsq(iz,:) = (z0 + dz*(iz-1))**2
23006        continue
c
c start of the main do-loop
c
      do 23012 iz= 1,nz
         do 23014 ix= 1,nx
c             'iz:nt' means: access all array elements from index iz to nt
              sma(iz:nt,ix) = (tsqm(iz:nt,ix) -zsq(iz:nt,ix))
     &                                         /xsq(iz:nt,ix)
              isma(iz:nt,ix)=1.5+(sma(iz:nt,ix)-s0)/ds
              spa(iz:nt,ix) = (tsqp(iz:nt,ix) -zsq(iz:nt,ix))
     &                                         /xsq(iz:nt,ix)
              ispa(iz:nt,ix)=1.5+(spa(iz:nt,ix)-s0)/ds
              where ( isma(iz:nt,ix).lt.2 .or. ispa(iz:nt,ix).gt.ns )
                  con(iz:nt, ix) = 0.0
              elsewhere
                  sca(iz:nt,ix)=sqrt(tt(iz:nt,ix)/(1.+real(ispa(iz:nt,ix)
     &                            -isma(iz:nt,ix))))/xx(iz:nt,ix)
                  con(iz:nt, ix) = tx(iz:nt ,ix) * sca(iz:nt,ix)
              end where
23014      continue
           do 23013 ix=1,nx     ! smearing along the slowness axis
              zs(iz,:) = zs(iz,:) + con(iz:nt,ix)
23013      continue
23012   continue
      return
      end
```

My second version, "Vspray2", reduces the communication considerably but adds some serialness. Input and output array are aligned with each other. Some table arrays are aligned to them in the same way as in the previous example. The offset and slowness dimensions are serial; i.e. they are stored in one virtual processor. Every time slice is copied to every processor memory so that for given τ each slowness value can be computed independently within each processor. The number of virtual processors is therefore equal to the number of time samples of the input data.

```
c Program Vspray2
c
c start of the main do-loop
c
      do 23016 it=1,nt
c             SPREAD() is an intrinsic CM Fortran function.
c             Here, each time slice tx(it,:) is spread along the time
c             axis (DIM=1) nt-times, and is stored in txhelp
c
              txhelp = SPREAD(tx(it,:),DIM=1,NCOPIES=nt)
              spa(it:nt,1:nx) = (tsqp(it:nt,1:nx) -zsq(it:nt,1:nx))
     &                                          /xsq(it:nt,1:nx)
              isma(it:nt,1:nx)=1.5+(sma(it:nt,1:nx)-s0a)/ds
              spa(it:nt,1:nx) = (tsqp(it:nt,1:nx) -zsq(it:nt,1:nx))
     &                                          /xsq(it:nt,1:nx)
              ispa(it:nt,1:nx)=1.5+(spa(it:nt,1:nx)-s0)/ds
              where ( isma(it:nt,1:nx).lt.2 .or. ispa(it:nt,1:nx).gt.ns )
                  con(it:nt, 1:nx) = 0.0
              elsewhere
                  sca(it:nt,1:nx)=sqrt(tt(it:nt,1:nx)/(1.+real(ispa(it:nt,1:nx)
     &                               -isma(it:nt,1:nx))))/xx(it:nt,1:nx)
                  con(it:nt, 1:nx) = txhelp(it:nt ,1:nx) * sca(it:nt,1:nx)
              end where
              do 23013 ix=1,nx           ! smearing along the slowness axis
                   zs(it, 1:ns) = zs(it, 1:ns) + con(it:nt,ix)
23013         continue
23016    continue
      return
      end
```

My final version, "Vspray3", is probably not optimal but is faster than the second program. Even the offset dimension is now parallel. The amount of interprocessor communication is not reduced by making the program more serial, but by applying the following trick: After computing the slowness and smearing along the slowness axis for each τ-slice, which is done parallel in the time and offset domains, the whole seismogram is shifted along its τ-axis. The result is a "systolic" data flow at the end of each loop over τ. This avoids the transmission of the next time slices necessary for computation of the next τ-slice and explains the improvement in performance compared to the last program.

```
c Program Vspray3
c
c  start of the main loop
c
      txhelp = tx       ! txhelp is not needed, if the input data may be changed
      do 23016 it=1,nt
```

```
            sma(it:nt,1:nx) = (tsqm(it:nt,1:nx) -zsq(it:nt,1:nx))
     &                                         /xsq(it:nt,1:nx)
            isma(it:nt,1:nx)=1.5+(sma(it:nt,1:nx)-s0)/ds
            spa(it:nt,1:nx) = (tsqp(it:nt,1:nx) -zsq(it:nt,1:nx))
     &                                         /xsq(it:nt,1:nx)
            ispa(it:nt,1:nx)=1.5+(spa(it:nt,1:nx)-s0)/ds
            where ( isma(it:nt,1:nx).lt.2 .or. ispa(it:nt,1:nx).gt.ns )
                con(it:nt, 1:nx) = 0.0
            elsewhere
                sca(it:nt,1:nx)=sqrt(tt(it:nt,1:nx)/(1.+real(ispa(it:nt,1:nx)
     &                      -isma(it:nt,1:nx))))/xx(it:nt,1:nx)
                con(it:nt, 1:nx) = txhelp(it:nt ,1:nx) * sca(it:nt,1:nx)
            end where
            do 23013 ix=1,nx          ! smearing over slowness
                 zs(it,1:ns) = zs(it,1:ns) + con(it:nx,ix)
23013       continue
c           EOSHIFT is an intrinsic CM Fortran function
c           Here, the array txhelp is left shifted along the time axis
c           about one sample and the rightmost arrayelement is set to zero
c
            txhelp = EOSHIFT(txhelp, DIM=2, SHIFT=1, BOUNDARY=0.0)
23016   continue
      return
      end
```

The performances of the three program versions differ considerably. The computing time is not a linear function of the length of the I/O-arrays as it is approximately for serial computers, when swapping can be avoided. This is not surprising because of the nonlinear influence of the communication pattern. To provide a more objective comparison, a vectorized version, “Vspray4”, was run on our Convex C1-XP. Computations made on that machine and on 8k processors on a 32k Connection Machine in Cambridge, Massachusetts, are compared in Table 1. The absolute CPU times shouldn't be taken too seriously because the actual run time depends partly on how many people are working on the front end.

Table 1.	Performance Comparison				
Parameters	Vspray1	Vspray2	Vspray3	Vspray4	
nt=200	361 sec	86 sec	48 sec	9 sec	run time
nx=20	1	1	1	1	X first row run time
ns=20	40	10	5	1	X Convex run time
nt=1000	11541 sec	2199 sec	1479 sec	564 sec	run time
nx=64	32	26	31	63	X first row run time
ns=64	20	4	2.6	1	X Convex run time
nt=2000	several hours	4118 sec	2930 sec	2237 sec	run time
nx=64	unknown	48	61	250	X first row run time
ns=64	unknown	2	1.3	1	X Convex run time
nt=8000	several days(?)	8815 sec	7633 sec	18943 sec	run time
nx=32	unkown	102	159	2104	X first row run time
ns=32	unknown	0.5	0.4	1	X Convex run time

Nevertheless the relative computing times are interesting. The table tells us that the advantage of using the CM generally increases with the size of the task, but turns to a disadvantage if the data set is too small. The parallelization is only effective if the parallel array dimensions are at least as large as the number of physical processors.

It should be mentioned that if processing many shotgathers –not only one, like I did– it is better to compute the slowness as function of τ, time and offset in advance, store it in a 3-dimensional array, and shift it over the shotgathers. In that case the performance advantage of the CM will increase rapidly.

The huge discrepancies in computing time suggest that the programmer should think very carefully about the optimal solution before he or she implements an algorithm. This is much more important in parallel than in serial computing.

CONCLUSION

To write a program in CM Fortran is easy because the language is simple. The aim of writing a fast program can only be achieved if the programmer rethinks his serial algorithm. The task is to find the optimal compromise between minimizing the interprocessor communication and maximizing the degree of parallelization. Crucial to achieving this goal is the assignment of the arrays to their homes on virtual processors. In addition, the effective use of the intrinsic CM functions to perform parallel instructions can increase the program's speed considerably.
Although the performance of the Connection Machine for the pixel-precise velocity scanning algorithm is relatively poor, I think a much faster velocity analysis is possible on the CM. The standard velocity analysis programs don't smear along the slowness axis and their implementations should be faster.

REFERENCES

Boghosian, B.M., 1990, Computational Physics on the Connection Machine: Computer in Physics, Jan/Feb 1990.

Claerbout, J.F., 1989, Preface to Practical Inversion Tutorial: SEP-**61**, p. 361-362.

Claerbout, J.F., 1990, Hyperbola Tricks: SEP-**65**.

Delany, H.C., 1988, Ray Tracing on the Connection Machine: Thinking Machines Corporation Technical Report VZ88-3.

Mora, P., 1989, Modeling Waves in the Earth: Thinking Machines Corporation Technical Report GEO89-1.

Hillis, W.D., 1985, The Connection Machine: The MIT Press, Cambridge, Massachusetts.

Long, L.N., 1989, A three-dimensional Navier-Stokes method for the Connection Machine: in Scientific Applications of the Connection Machine, World Scientific Publishing, Singapore.

Myers, D.W., and Adams III, G.B., 1989, Benchmarking and performance analysis of the CM-2: in Scientific Applications of the Connection Machine, World Scientific Publishing, Singapore.