SSRUN and PROF are tools from SGI. It is often useful to put timing routine by yourself to track the performance of a particular loop or routine.

The following table shows the common timing functions available for F77 on the SGI:

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>Return the number of seconds elapsed since 1 January, 1970</td>
</tr>
<tr>
<td>itime</td>
<td>Return the current hour, minute, and second in numerical array,</td>
</tr>
<tr>
<td>ctime</td>
<td>Convert the time returned by the time function to a character string,</td>
</tr>
<tr>
<td>ltime</td>
<td>Convert the time returned by the time function to the local time,</td>
</tr>
<tr>
<td>gmtime</td>
<td>Convert the time returned by the time function to Greenwich time,</td>
</tr>
<tr>
<td>etime</td>
<td>Return the elapsed user and system execution time since last call to etime,</td>
</tr>
<tr>
<td>dtime</td>
<td>Return the elapsed user and system execution time since last call to dtime</td>
</tr>
<tr>
<td>second</td>
<td>Returns elapsed CPU time (in second) since last call to second,</td>
</tr>
<tr>
<td>timef</td>
<td>Returns elapsed wall-clock time in milliseconds since the previous call to timef</td>
</tr>
</tbody>
</table>

Amount the above functions, the last 4 may be most useful in order to time your code.
Consider the following example:

```fortran
PROGRAM time
 c     Integrate the Sine function from 0 to pi by Simpson Rule.
 c
 implicit none
 real f(90000),pi,sim,theta
 integer n,i
 c
 real*4 etime, dtime, second
 real*4 tarray(2)
 real timef
 real time_timef
 c
 n = 90000
 pi = 4.*atan(1.)
 c
 Set up function and dimension
 do i=1,n
   theta = float(i-1)*pi/float(n-1)
   f(i) = sin(theta)
 enddo
 c
 time_timef = timef()
 do i=1,1000
   call SIMPSON(f,n,0.,pi,sim)
 enddo
 time_timef = timef()
 c
 write(*,*)'Timing from TIMEF()', time_timef
 stop
end
```

The above example uses TIMEF() to track the computational time that used by Simpson Rule. Similar usage for dtime but need to declare as real*4.

For etime and second, it should be something like:

```fortran
begin_time = second()
 do i=1,1000
   call SIMPSON(f,n,0.,pi,sim)
 enddo
 end_time = second()
 write(*,*)'Timing from second',end_time-begin_time
```

Below we will talk about the various timing routines individually.
timef Functions

This function comes from CRAY library. It gives time in msec. If using f77, -craylibs flag is need:

```
f77 -r8 -64 -o time time.f -craylibs
```

However, you don't need -craylibs when using f90 compiler. If you don't use -r8 flag, you have to declare timef to be real*8. Otherwise you get wrong timing.

For example:

```
f77 -r8 -64 -o time time.f -craylibs
```

| SIMPSON Rule Result: | 1.999977777574699 |
| Timing from TIMEF()  | 4547.5871999999999 |

```
f77 -64 -o time time.f -craylibs
```

| SIMPSON Rule Result: | 1.99998403 |
| Timing from TIMEF()  | -3.778950758E-14 (WRONG!) |

etime and dtime Functions

This function gives time in seconds. For example:

```
f77 -64 -o time time.f
```

| SIMPSON Rule Result: | 1.999984 |
| Timing from ETIME()  | 4.241411 |

```
f77 -r8 -64 -o time time.f        (etime and tarray(2) declared as real, not real*4)
```

| SIMPSON Rule Result: | 1.999977777574699 |
| Timing from ETIME()  | 0.0000000000000000E+00 (WRONG!) |

second Functions

This function gives time in sec. and needs SGI library complib.sgimath. For example:

```
f77 -r8 -64 -o time.f -lcomplib.sgimath
```

| SIMPSON Rule Result: | 1.999977777574699 |
| Timing from SECOND() | 4.513525970280170 |

```
f77 -r8 -64 -o time.f        (second declared as real, not real*4)
```

| SIMPSON Rule Result: | 1.999977777574699 |
| Timing from ETIME()  | 0.0000000000000000E+00 (WRONG!) |
CHAPTER 2  

Tricks

*Multiplication verses Division*

The basic idea is to replace division inside a do-loop by multiplication. Division instruction is about 100 times longer than multiplication.

The example below* shows the original code has division by SIGMA and Q2. Create SIGMAi and Q2inv will help reduce the computational time.

* The code, which is used to solve the semiconductor Bloch equations, is provided by Dr. Cun-Zheng Ning NASA Ames Research Center.
Example

The original code:

SUBROUTINE SCATT(KX,KY,KZ,FSC,CRHOSC)
  IMPLICIT REAL*8 (A-B,D-H,O-Z)
  IMPLICIT COMPLEX*16 (C)
  DIMENSION DK(0:11),FSC(2)
  COMMON/NMAX2/DNMAX2
  COMMON/NMAX3/DNMAX3
  SIGMA=0.5D0/DNMAX2
  DO 10 IZ=0,N
  DO 10 IY=0,N
  DO 10 IX=0,N
    DE=DBLE(KXAB+KYAB+KZAB-IX-IY-IZ)/DNMAX2
    E1=(DE-OME0)/SIGMA !flag
    C--- delta function ---------------
    DELTA=EXP(-E1*E1)/SIGMA !flag
    C---------------------------------
  11  TMP=0.D0
    ...
    ...
    QX2=QX*QX
    QY2=QY*QY
    QZ2=QZ*QZ
    Q2=QX2+QY2+QZ2
    IF(Q2.GT.1.D-10) THEN
      IF(IFLAG.EQ.1) THEN
        TMP=FSCAT1(KX,KY,KZ,IX,IY,IZ,1)/Q2
        TMP2=FSCAT1(KX,KY,KZ,IX,IY,IZ,2)/Q2
        CTMP=CRHOS1(KX,KY,KZ,IX,IY,IZ)/Q2
        ENDF
        Q1=SQRT(Q2)
        ...
        (At least 16 more of this type of blocking)
      ENDIF
    ELSE
      ...
    ENDIF
  END

Timing (in second) for 10 CPUs on turing : (one iteration)

Original:
  RUNGE: 200.744278
Modified:
  RUNGE: 176.407455

For this case, saving 24 seconds per time step is very significant.

The modified code:

SUBROUTINE SCATT(KX,KY,KZ,FSC,CRHOSC)
  IMPLICIT REAL*8 (A-B,D-H,O-Z)
  IMPLICIT COMPLEX*16 (C)
  DIMENSION DK(0:11),FSC(2)
  COMMON/NMAX2/DNMAX2
  COMMON/NMAX3/DNMAX3
  SIGMA=0.5D0*DNMAX2i !CHEUNG
  DO 10 IZ=0,N
  DO 10 IY=0,N
  DO 10 IX=0,N
    DE=DBLE(KXAB+KYAB+KZAB-IX-IY-IZ)*DNMAX2i
    E1=(DE-OME0)*SIGMAi !flag CHEUNG
    C--- delta function ---------------
    DELTA=EXP(-E1*E1)*SIGMAi !flag CHEUNG
    C---------------------------------
  11 TMP=0.D0
    ...
    ...
    QX2=QX*QX
    QY2=QY*QY
    QZ2=QZ*QZ
    Q2=QX2+QY2+QZ2
    Q2inv = 1./Q2 !CHEUNG
    IF(Q2.GT.1.D-10) THEN
      IF(IFLAG.EQ.1) THEN
        TMP=FSCAT1(KX,KY,KZ,IX,IY,IZ,1)*Q2inv
        TMP2=FSCAT1(KX,KY,KZ,IX,IY,IZ,2)*Q2inv
        CTMP=CRHOS1(KX,KY,KZ,IX,IY,IZ)*Q2inv
        ENDF
        Q1=SQRT(Q2)
        ...
        (At least 16 more of this type of blocking)
      ENDIF
    ELSE
      ...
    ENDIF
  END
**Stride Penalty**

Every loop should use a memory stride of 1; that is, a loop over an array should access array elements from adjacent memory addresses. When the loop "walks" through memory by consecutive words, it uses every word of every cache line in sequence, and does not return to any cache line after finishing it.

Consider the loop nest in this example.

<table>
<thead>
<tr>
<th>Poor Cache Use</th>
<th>Achieve Stride-One Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>do i = 1, n</td>
<td>do i = 1, n</td>
</tr>
<tr>
<td>do j = 1, n</td>
<td>do j = 1, n</td>
</tr>
<tr>
<td>a(i,j) = b(i,j)</td>
<td>a(i,j) = b(i,j)</td>
</tr>
<tr>
<td>enddo</td>
<td>enddo</td>
</tr>
<tr>
<td>enddo</td>
<td>enddo</td>
</tr>
</tbody>
</table>

In Fortran, this means copying adjacent memory locations in sequence. Thus, all data in the same cache line are used, and each line needs to be loaded only once.

The following example* shows the original code suffered from a lot of L1 and L2 data cache misses. The two lines of code where causing problems are 118 and 119. This is due to stride penalty (not a stride-one access).

In addition, the original code has a long L-loop which can’t be parallelized right away!

One can kill two birds in one stone by storing the pkmin and pkmax into a vector of 2 dimensions. This allows easy parallelization and get rid of the stride penalty coming from fetching pkz(i,1,L) and pkz(i,1,L+1) at the same time.

The final code looks very much the same as the original one. See the comparison below:

* From DAO GCM siggtop routine.
Example

### The Original Code

c Interpolate Pressure Between Sigma Levels

do L=1,lm-1
   pkmin = pkz(1,1,L)
   pkmax = pkz(1,1,L+1)
   do i=2,im*jm
      if( pkz(i,1,L+1) .lt.pkmin )
         pkmin = pkz(i,1,L)      line 118
      if( pkz(i,1,L+1).gt.pkmax )
         pkmax = pkz(i,1,L+1)    line 119
   enddo
   if ( pk.le.pkmax .and. pk.ge.pkmin ) then
      do i=1,im*jm
         if ( pk.le.pkz(i,1,L+1) .and.
             pk.ge.pkz(i,1,L) ) then
            temp = ( pkz(i,1,L)-pk ) /
                ( pkz(i,1,L)-pkz(i,1,L+1) )
            if( qsig(i,1,L) .ne.undef .and.
                qsig(i,1,L+1).ne.undef ) then
               qprs(i,1) = qsig(i,1,L+1)*temp +
                qsig(i,1,L)*(1.-temp)
            else if( qsig(i,1,L+1).ne.undef .and.
                temp.ge.0.5 ) then
               qprs(i,1) = qsig(i,1,L+1)
            else if( qsig(i,1,L).ne.undef .and.
                temp.le.0.5 ) then
               qprs(i,1) = qsig(i,1,L)
         endif
      enddo
   endif
endo
endo
endo

### Modified Code

c$doacross local(L,i)
do L=1,lm-1
   pkmx(1,L) = pkz(1,1,L)
   pkmx(2,L) = pkz(1,1,L)
   do j=1,jm
      do i=1,im
         if( pkz(i,j,L) .lt.pkmx(1,L) )
            pkmx(1,L) = pkz(i,j,L)
         if( pkz(i,j,L).gt.pkmx(2,L) )
            pkmx(2,L) = pkz(i,j,L)
      enddo
   enddo
   c$doacross
   c$& local(L,i,temp),shared(qprs),
c$& mp_schedtype=dynamics
   do L=1,lm-1
      if( pk.le.pkmx(2,L+1) .and.
          pk.ge.pkmx(1,L) ) then
         do j=1,jm
            do i=1,im
               if( pkz(i,j,L+1) .lt.pkz(i,j,L) ) then
                  temp = ( pkz(i,j,L)-pk ) /
                      ( pkz(i,j,L)-pkz(i,j,L+1) )
               if( qsig(i,j,L) .ne.undef .and.
                  qsig(i,j,L+1).ne.undef ) then
                  qprs(i,j) = qsig(i,j,L+1)*temp +
                    qsig(i,j,L)*(1.-temp)
               else if( qsig(i,j,L+1).ne.undef .and.
                  temp.ge.0.5 ) then
                  qprs(i,j) = qsig(i,j,L+1)
               else if( qsig(i,j,L) .ne.undef .and.
                  temp.le.0.5 ) then
                  qprs(i,j) = qsig(i,j,L)
            endif
         enddo
      endif
      enddo
endo
endo
endo
endo
**Cache Utilization**

The operation of the caches immediately leads to some straightforward principles. Two of the most obvious are:

- A program ought to make use of every word of every cache line that it touches.

  Clearly, if a program does not make use of every word in a cache line, the time spent loading the unused parts of the line is wasted.

- A program should use a cache line intensively and then not return to it later.

  When a program visits a cache line a second time after a delay, the cache line may have been displaced by other data, so the program is delayed twice waiting for the same data.

Thus, one should:

1. Keep everything in cache
2. Put things in one do-loop

The following code* gives poor scaling performance on an Origin 2K

<table>
<thead>
<tr>
<th># of CPUs</th>
<th>Time (msec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>49.2588043</td>
</tr>
<tr>
<td>4</td>
<td>38.3873482</td>
</tr>
<tr>
<td>8</td>
<td>35.6569061</td>
</tr>
</tbody>
</table>

In the second do-loop the original code has ple() and pkle() assigned; then used them in the last loop when calculating pkz(). Each the second and the third loop has a very bad stride penalty. Elimination of these vectors, ple() and pkle() reduces the cache misses and increase flop rate.

Note that the function ple() is replaced by ple1 and ple2. The vector pkle() is 2-dimensional instead of 3 dimensional. There is no stride penalty on L.

* From DAO GCM pkappa() routine
Original Code

```
    do j = 1, jm
    do i = 1, im
        pinv(i,j) = 1.0 / ( akap1 *pz(i,j) )
        ple(i,j,1) = sige(1)*pz(i,j)+ptop
        pkle(i,j,1) = ( sige(1)*pz(i,j)+ptop )**akap
        enddo
    enddo

    c$doacross local (i,j,l)
    do L = 2, lm+1
        do j = 1, jm
            do i = 1, im
                ple(i,j,L) = sige(L)*pz(i,j)+ptop
                pkle(i,j,L) = pkht(i,j,L-1)
            enddo
        enddo
    enddo

    c$doacross local (i,j,l,dsiginv)
    do L = 1, lm
        dsiginv = 1.0/dsig(L)
        do j = 1, jm
            do i = 1, im
                pkz(i,j,L) = ( ple(i,j,l+1)*pkle(i,j,l+1) &
                               - ple(i,j,l )*pkle(i,j,l )
                             & * pinv(i,j)*dsiginv
            enddo
        enddo
    enddo
```

Modified Code

```
    do j = 1, jm
    do i = 1, im
        pinv(i,j) = 1.0 / ( akap1 *pz(i,j) )
        ple(i,j,1) = sige(1)*pz(i,j)+ptop
        pkle(i,j) = ( sige(1)*pz(i,j)+ptop )**akap
        enddo
    enddo

    c For L=1
    L = 1
    dsiginv = 1.0/dsig(L)
    c$doacross local (i,j,ple1,ple2)
    do j = 1, jm
        do i = 1, im
            ple1 = sige(1)*pz(i,j)+ptop
            ple2 = sige(2)*pz(i,j)+ptop
            pkz(i,j,1) = ((ple2*pkht(i,j,1) - ple1*pkle(i,j)) &
                                                  * pinv(i,j)*dsiginv
        enddo
    enddo

    c$doacross local (i,j,l,dsiginv,plel,plelp)
    do L = 1, lm
        dsiginv = 1.0/dsig(L)
        do j = 1, jm
            do i = 1, im
                plel = sige(L)*pz(i,j)+ptop
                plelp = sige(L+1)*pz(i,j)+ptop
                pkz(i,j,L) = ( plel*pkht(i,j,L) - plelp*pkht(i,j,L) &
                                             * pinv(i,j)*dsiginv
            enddo
        enddo
    enddo
```

Parallel Computation
One has to identify the fact that the third loop is calculating pkz() which needs pkle(). We have stride penalty from pkle(). Since pkle() is obtained from pkht(); it makes sense to obtain pkz() from pkht() directly from the third loop.

Performance on the Modified code:

<table>
<thead>
<tr>
<th># of CPUs</th>
<th>Time (msec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>47.1265678</td>
</tr>
<tr>
<td>4</td>
<td>24.0419426</td>
</tr>
<tr>
<td>8</td>
<td>12.5438089</td>
</tr>
<tr>
<td>16</td>
<td>6.72540569</td>
</tr>
<tr>
<td>24</td>
<td>4.88756609</td>
</tr>
</tbody>
</table>

**Vector Copying I**

This is a very powerful technique to reduce cache misses and improve performance. The idea is to make copies of the addressing vectors so that the local vectors are local in cache. But also note that vector copying also needs time and memory. Thus this technique works only in those routines are being called many times.

Consider a segment of a code* below with bad secondary caches misses. The first obvious thing is that the original code has very bad stride penalty. The k-loop should be outside or the vectors should have k in the inner index. For this code, changing all the indexes is way too complicated. However, one cannot simply move the k-loop to outside. The original code has yz() calculated for each i and j. If we store the value of yz() for each i and j, this will make things easier. Create a new vector yz3() which is the original yz() vector with i, j, and k dimensions stored. By doing this, one can put the k-loop outside and reduce the cache misses, see the modified code. Note that there are two outer loops.

*omyz() routine from SWIFT by Rodrick V. Chima at NASA Lewis
One common CFD practice is sweeping through different directions. If the indexing is (i,j,k); when sweeping in the i-direction, you can have the i-loop inside; thus no cache misses via stride penalty. However, when sweeping the k-direction, we normally have the k-loop as the most inner loop; thus the code performance is hurt by secondary cache misses.

Consider the following code segment*:
!$OMP DO
   do 360  j=2,j1,1
      do 330  i=2,i1,1
         c(1)=0.
         c(km)=0.
         do 301  l=1,5,1
df(l,1)=0.
      301         df(l,k1)=0.
      do 305  k=2,k1,1
         sum=p(i,j,k-1)+p(i,j,k+1)
         c(k)=abs(sum-2.*p(i,j,k))/min(pdmax,abs(sum+2.*p(i,j,k)))
   305         b(k)=ep(3,i,j,k)/rj(i,j,k)
      do 310  k=2,k2,1
         sum=p(i,j,k-1)+p(i,j,k+1)
         c(k)=abs(sum-2.*p(i,j,k))/min(pdmax,abs(sum+2.*p(i,j,k)))
   310         b(k)=ep(3,i,j,k)/rj(i,j,k)
      CONTINUE
      do 315  k=1,k1,1
      do 315  l=1,5,1
df(l,k)=a(k)*dq(l,k)-b(k)*(dq(l,k-1)-2.*dq(l,k)+dq(l,k+1))
   315         CONTINUE
   360     CONTINUE
!$OMP END DO nowait
*---------------------------------------------------
*     multiply df by inverse preconditioning matrix
*---------------------------------------------------
if (ipc.gt.0) THEN
   do 325  k=2,k2,1
      rho=gdgm*p(i,j,k)/qq(5,i,j,k)
      u=qq(2,i,j,k)
      v=qq(3,i,j,k)
      w=qq(4,i,j,k)
      vq=u**2+v**2+w**2
      vdp=u**2+(v-om*z(i,j,k))**2+(w+om*y(i,j,k))**2
      bmr=min(max(vdp,refv),gm*qq(5,i,j,k))
      cr5=qq(5,i,j,k)+.5*vq-bmr
      df1=df(1,k)/bmr
      df2=df(2,k)
      df3=df(3,k)
      df4=df(4,k)
      df(1,k)=df1
df(2,k)=df1*u+df2*rho
df(3,k)=df1*v+df3*rho
df(4,k)=df1*w+df4*rho
df(5,k)=df1*cr5+rho*(u*df2+v*df3+w*df4+df(5,k))
   325     CONTINUE
endif
   do 330  k=2,k1,1
   do 330  l=1,5,1
      av(l,i,j,k)=av(l,i,j,k)+df(l,k)-df(l,k-1)
   330     CONTINUE
!$OMP END DO nowait
As seen in the above example, the variables have index of \((i,j,k)\). For example, \(v(5,i,j,k)\), \(qq(4,i,j,k)\), and \(y(i,j,k)\) etc. These variables are suffering secondary caches misses, revealed by SSRUN.

We divide the code into 2 parts, one is before “multiply \(df\) by inverse preconditioning matrix” and one is after. New and bigger vectors \(df6(), b3(), c3()\) is used to replace \(df(), b()\) and \(c()\) respectively. The code below is the modification of the original. Note that the loops with the same loop number are doing the same job in both original and modified codes.

\[
\begin{align*}
!$OMP DO \\
do & j=1,j1,1 \\
do & i=1,i1,1 \\
c3(i,j,l) = 0. \\
c3(i,j,km) = 0. \\
do & l=1,5 \\
df6(l,i,j,1) = 0. \\
df6(l,i,j,k1) = 0. \\
\end{align*}
\]

\[
\begin{align*}
!$OMP END DO \\
!$OMP DO \\
do & j=2,j1,1 \\
do & i=2,i1,1 \\
sum=p(i,j,k-1)+p(i,j,k+1) \\
c3(i,j,k)=abs(sum-2.*p(i,j,k))/min(pdmax,abs(sum+2.*p(i,j,k))) \\
b3(i,j,k)=ep(3,i,j,k)/rj(i,j,k) \\
\end{align*}
\]

\[
\begin{align*}
!$OMP END DO \\
!$OMP DO \\
do & k=2,k1,1 \\
do & j=2,j1,1 \\
do & i=2,i1,1 \\
bb=.5*(b3(i,j,k)+b3(i,j,k+1)) \\
v2=av1+av2*max(c3(i,j,k-1),c3(i,j,k),c3(i,j,k+1),c3(i,j,k+2)) \\
v4=max(0.,av4-v2) \\
fk=min(1.,rkh*(.5+float(k)),rkt*(.5+float(kp-k))) \\
a3(i,j,k)=v2*bb \\
b3(i,j,k)=v4*bb*fk \\
\end{align*}
\]

\[
\begin{align*}
!$OMP END DO \\
!$OMP DO \\
do & k=2,k2,1 \\
do & j=2,j1,1 \\
do & i=2,i1,1 \\
dqlk = qq(1,i,j,k+1)-qq(1,i,j,k) \\
dqlkp1 = qq(1,i,j,k+2)-qq(1,i,j,k+1) \\
dqlkm1 = qq(1,i,j,k)-qq(1,i,j,k-1) \\
df6(1,i,j,k)=a3(i,j,k)*dqlk- \\
& b3(i,j,k)*( dqlkm1 - 2.*dqlk + dqlkp1)
\end{align*}
\]
Tricks

enddo
enddo
enddo

320 continue
!$OMP END DO

Perfomance on the Modified routine (1 CPU):

<table>
<thead>
<tr>
<th>Run #</th>
<th>Original (msec.)</th>
<th>Modified (msec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.10804892</td>
<td>1.36329401</td>
</tr>
<tr>
<td>2</td>
<td>2.12119484</td>
<td>1.35878205</td>
</tr>
</tbody>
</table>

There is a significant improvement in speed when reducing secondary cache misses.
Cache Thrashing

Consider the code fragment below, where three large vectors are combined into a fourth.

```fortran
parameter (max = 1024*1024)
dimension a(max), b(max), c(max), d(max)
do i = 1, max
   a(i) = b(i) + c(i)*d(i)
endo d = 0.0
```

The four vectors are declared one after the other, so they are allocated contiguously in memory. Each vector is 4 MB in size (1024*1024 times the size of a default real, 4 bytes). Thus the low 22 bits of the addresses of elements a(i), b(i), c(i), and d(i) are the same. The vectors map to the same locations in the associative caches.

To perform the calculation in iteration i of this loop, a(i), b(i), c(i), and d(i) must be resident in the primary cache. c(i) and d(i) are needed first to carry out the multiplication. They map to the same location in the cache, but both can be resident simultaneously because the cache is two-way set associative - it can tolerate two lines using the same address bits. To carry out the addition, b(i) needs to be in the cache. It maps to the same cache location, so the line containing c(i) is displaced to make room for it (here we assume that c(i) was accessed least recently). To store the result in a(i), the cache line holding d(i) must be displaced.

Now the loop proceeds to iteration i+1. The line containing c(i+1) must be loaded anew into the cache, because one of the following things is true:

- c(i+1) is in a different line than c(i) and so its line must be loaded for the first time.
- c(i+1) is in the same line as c(i) but was displaced from the cache during the previous iteration.

Similarly, the cache line holding d(i+1) must also be reloaded. In fact, every reference to a vector element results in a cache miss, because only two of the four values needed during each iteration can reside in the cache simultaneously. Even though the accesses are stride-one, there is no cache line reuse.

This behavior is known as cache thrashing, and it results in very poor performance, essentially reducing the program to uncached use of memory. The cause of the thrashing is the unfortunate alignment of the vectors: they all map to the same cache location.
Using Array Padding to Prevent Thrashing

There are ways to repair cache thrashing:

1. Redimension the vectors so that their size is not a power of two. A new size that spaces the vectors out in memory so that a(1), b(1), c(1) and d(1) all map to different locations in the cache is ideal. For example, max = 1024 * 1024 + 32 would offset the beginning of each vector 32 elements or 128 bytes. This is the size of an L2 cache line, so each vector begins at a different cache address. All four values may now reside in the cache simultaneously, and complete cache line reuse is possible.

2. Introduce padding variables between the vectors in order to space out their beginning addresses. Ideally, each padding variable should be at least the size of a full cache line. Thus if max is kept the same, the following declaration eliminates cache thrashing:

   \[
   \text{dimension a(max), pad1(32), b(max), pad2(32), } \\
   \text{& c(max), pad3(32), d(max)}
   \]

3. For multidimensional arrays, it is sufficient to make the leading dimension an odd number, as in the following:

   \[
   \text{dimension a(1024+1,1024)}
   \]

4. For arrays with smaller dimensions, it is necessary to change two or more dimensions, as in the following:

   \[
   \text{dimension a(64+1,64+1,64)}
   \]

Eliminating cache thrashing makes the loop at least 100 times faster.
Using Prefetching

prefetch instructions move data from main memory into cache in advance of their use. To see how this works, consider the simple reduction loop

```fortran
  do i=1,n
    a = a + b(i)
  enddo
```

If the loop is executed exactly as written, every secondary cache miss of the vector b stalls the CPU, because there is no other work to do. Now consider the modification of the above example. In this example, "prefetch" indicates only that a prefetch instruction for the data at the specified address is issued at that point in the program.

```fortran
  do i = 1, n
    c*$* prefetch_ref=b(i+16)
    a = a + b(i)
  enddo
```

If b is an array of real*8, the address b(i+16) is one L2 cache line (128 bytes) ahead of the value that is being read in the current iteration. With this prefetch instruction inserted in the loop, each cache line is requested 16 iterations before it needs to be used. Each iteration of the loop takes two cycles: the prefetch instruction and the load of b(i), each take one cycle and the addition is overlapped. Therefore, cache lines are prefetched 32 cycles (16, 2-cycle iterations) in advance of their use. The latency of a cache miss to local memory is approximately 60 cycles, and roughly half of this time will be overlapped with work on the previous cache line.

Compiler options for prefetching are very handy:

- `-LNO:prefn= n` - Enable or disable prefetching for cache level n (n=1,2,3,4)
- `-LNO:prefetch=n` - Disables all prefetching
- `-LNO:prefetch__ahead=n` - How many cache lines ahead to prefetch. (default n=2)
Adjusting Cache Blocking Block Sizes

Understanding Cache Blocking

Here, data structures that are too big to fit in the cache are broken up into smaller pieces that will fit in the cache. Consider the matrix multiplication code

\[
\begin{align*}
&\text{do } j = 1, n \\
&\quad \text{do } i = 1, m \\
&\quad \quad \text{do } k = 1, l \\
&\quad \quad \quad c(i, j) = c(i, j) + a(i, k) \times b(k, j) \\
&\quad \end{aligned}
\]

If arrays a, b, and c are small enough that all fit in cache, performance is great. But if they are too big, performance drops substantially.

<table>
<thead>
<tr>
<th>m</th>
<th>n</th>
<th>p</th>
<th>Seconds</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>30</td>
<td>30</td>
<td>0.000162</td>
<td>333.9</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>200</td>
<td>0.056613</td>
<td>282.6</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>25.43118</td>
<td>78.6</td>
</tr>
</tbody>
</table>

Calculating one element of c requires reading an entire row of a and an entire column from b. If a and b don’t fit in the cache, the earlier rows and columns are likely to be displaced by the later ones, so there is likely to be little reuse.
Using Tuned Libraries

The quickest and easiest way to improve a program’s performance is to link it with libraries already tuned for the target hardware. The standard math library is so tuned, and there are optional libraries that can provide performance benefits: lib-fastm, CHALLENGEcomplib, and SCSL.

Standard Math Library

For the standard libraries such as libc, hardware-specific versions are automatically linked, based on the compiler’s information about the target system.

The standard math library includes special "vector intrinsics," that is, vectorized versions of the functions vacos, vasin, vatan, vcos, vexp, vlog, vsin, vsqrt, and vtan (plus single-precision versions whose names are made by appending an f). These functions are designed to take maximum advantage of the pipelining characteristics of the CPU when processing a vector of numbers.

The standard math library is described in the math(3) reference page. This library is linked in automatically by Fortran. When you link using the ld or cc commands, use the -lm flag.

libfastm Library

The libfastm library provides highly optimized versions of a subset of the routines found in the standard libm math library. Optimized scalar versions of sin, cos, tan, exp, log, and pow for both single and double precision are included. To link libfastm, append -lfastm to the end of the link line: cc -o modules...-lfastm.

Separate versions of libfastm exist for the R10000, R8000, and R5000. When the target system is not the one used for compiling, explicitly specify the target CPU. For details, see the libfastm(3) reference page.

CHALLENGEcomplib Library

CHALLENGEcomplib is a library of mathematical routines that carries out linear algebra operations, Fast Fourier Transforms (FFTs), and convolutions. It contains implementations of the well-known public domain libraries LINPACK, EISPACK, LAPACK, FFTPACK, and the Level-1, -2, and -3 BLAS, but all tuned specifically for high performance on the MIPS IV ISA.
Tricks

To link the sequential version, add -lcomplib.sgimath to the link line:

```bash
f77 -o modules -lcomplib.sgimath -lfastm
```

To link the parallel version, use -lcomplib.sgimath_mp; in addition, the -mp flag must be used when linking in the parallel code:

```bash
f77 -mp -o modules -lcomplib.sgimath_mp -lfastm
```

SCSL Library

SCSL, the Cray Scientific Library, is an optimized library that will eventually replace both CHALLENGEcomplib and Cray's libsci library. SCSL version 1.0 contains FFT routines, Level-1, -2, and -3 BLAS, and LAPACK routines, many of them parallelized.

You link SCSL into your program by using -lscs for the single-threaded version, or -lscs_mp for the parallelized version of the library.

Tip: Both CHALLENGEcomplib and SCSL define names also found in libfastm, so if you want to call libfastm it should be named last in the link command.