Advanced Shared Memory Programming

Ruud van der Pas

SiliconGraphics

European High Performance Computing Team

Topics

- *Automatic Parallelisation*
- *Manual Parallelisation*
- *Case Studies*
- *Appendix*

Automatic Parallelisation

Controlling Parallelism

The compilers attempt to parallelise the program:

- automatically (use the appropriate flags)

and

- as efficiently as possible (targetting the highest parallel level)

If this mechanism:

- fails and/or
- does not produce the desired results

a wealth of possibilities is available to manually control the parallelisation process

PFA/PCA – A New Begin

- Before release 7.2, auto–parallelisation was performed by a third party preprocesor
- With the 7.2 release this preprocessor is no longer used
- Instead, SGI proprietary compiler technology is now used for auto–parallelisation (at the WHIRL level)
- This allows for tighter integration with other optimisations and data distribution techniques

The Automatically Parallelising Option

- The APO is invoked through the –pfa option
- For all languages supported (F77,F90,C and C++) the compiler technology is the same
- Language differences may, and probably will, cause differences in compiler behaviour
- With data dependency analysis, loop (nests) are analyzed for automatic parallelization
- Directives/pragmas are inserted where it is determined to be safe to do so
- These directives/pragmas are then translated to the appropriate calls to MP library routines

Parallel Fortran and C – As of 7.2

Parallelising	Fortran	:	Power Fortran
Parallelising	C	:	Power C

Invocation is through a special flag:

Automatic	–pfa –pca	Fortran and C Obsoleted, aliased to –pfa
Manual	–mp	Fortran and C
Listings	–pfa keep	Fortran .l , .m and .anl files C .l and .w2c.c files
	–pfa –F/CLIST=on –mplist	Gives w2f.f or w2c.c file only Fortran and C, w2f.f/w2c.c before the LNO phase

PFA – *Example/1*

```
%f77 -pfa keep -c -O3 -n32 -mips4 MXM.f
```

```
      SUBROUTINE MXM (A, B, C, L, M, N)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION A(L,M), B(M,N), C(L,N)

      DO 100 K = 1, N
        DO 100 I = 1, L
          C(I,K) = 0.
100     CONTINUE
      DO 110 J = 1, M, 4
        DO 110 K = 1, N
          DO 110 I = 1, L
            C(I,K) = C(I,K) + A(I,J) * B(J,K)
     $              + A(I,J+1) * B(J+1,K) + A(I,J+2) * B(J+2,K)
     $              + A(I,J+3) * B(J+3,K)
110   CONTINUE
C
      RETURN
      END
```

The Original Source

PFA – *Example/2*

The MXM.l file:

```
Parallelization Log for Subprogram mxm_

    9: PARALLEL (Auto) __mpdo_mxm_1
   10: Not Parallel
          Loop is contained within a parallel loop.

   14: PARALLEL (Auto) __mpdo_mxm_2
   13: Not Parallel
          Loop is contained within a parallel loop.
   15: Not Parallel
          Loop is contained within a parallel loop.
```


PFA – *Example/3*

Fragment of the MXM.m file:

```
C$SGI start 1
        SUBROUTINE mxm(A, B, C, L, M, N)

         ... declarations ommitted .....
C
C       **** statements ****
C
C       DOACROSS will be converted to SUBROUTINE __mpdo_mxm_1
C$SGI start 2
C$DOACROSS if(((DBLE(__mp_sug_numthreads_func$()) *((DBLE(
C$& __mp_sug_numthreads_func$()) * 1.23D+02) + 2.6D+03)) .LT.((DBLE((
C$& __mp_sug_numthreads_func$() + -1)) *(DBLE(L) * DBLE(N))) * 3.0D00))
C$& ), local(K, I), shared(N, L, C)
        DO K = 1, N, 1
C$SGI start 3
          DO I = 1, L, 1
            C(I, K) = 0.0D00
          END DO
C$SGI end 3
        END DO
C$SGI end 2
C       DOACROSS will be converted to SUBROUTINE __mpdo_mxm_2
C$SGI start 5
C$DOACROSS local(K0, do_ivar, I0), shared(C, A, B, N, L, M)
        DO K0 = 1, N, 1
C$SGI start 4
          DO do_ivar = 0, (INTRN_I4DIVCEIL(M, 4) + -1), 1
C$SGI start 6
            DO I0 = 1, L, 1
              C(I0, K0) = ((((C(I0, K0) +(A(I0, (do_ivar * 4) + 1) * B((do_ivar * 4) + 1, K0)))
+(A(I0, (do_ivar * 4) + 2) * B((do_ivar * 4) + 2, K0))) +(A(I0, (do_ivar * 4) + 3) * B((do_ivar * 4)
+ 3, K0))) +(A(I0, (do_ivar * 4) + 4) * B((do_ivar * 4) + 4, K0)))
            END DO
C$SGI end 6
          END DO
C$SGI end 4
        END DO
C$SGI end 5
        RETURN
        END ! mxm
```


PFA – *Example/4*

Fragment of the MXM.anl file:

```
TRANSFORMATION_LOG_BEGIN
INTERCHANGE         4,5 5,4
TRANSFORMATION_LOG_END

PARALLELIZATION_LOG_BEGIN
tid 2 PARALLELIZED
tid 3 SERIAL_INSIDE_PARALLEL
tid 5 PARALLELIZED
tid 4 SERIAL_INSIDE_PARALLEL
tid 6 SERIAL_INSIDE_PARALLEL
PARALLELIZATION_LOG_END

DOACROSS_LOG_BEGIN
tid 2 "__mpdo_mxm_1"
tid 5 "__mpdo_mxm_2"
DOACROSS_LOG_END

NEST_LOG_BEGIN
tid 2 0
tid 3 1
tid 5 0
tid 4 1
tid 6 2
NEST_LOG_END


msg loop lines [1 9 7] "nesting depth: 2, estimated iterations: 11"
"Unrolled 8 times"

msg swps lines [1 9 7] "steady state"
"   12 estimated iterations before pipelining"
"    8 unrollings before pipelining"
"    8 cycles per 8 iterations"
"    8 mem refs     (100% of peak)"
"    2 integer ops  ( 12% of peak)"
"   10 instructions ( 31% of peak)"
"    1 short trip threshold"
"    2 integer registers used."
"    1 float register used."
```


PFA – *Example/4*

```
%f77 -n32 -mips4 -O3 -pfa -mplist MXM.f

translates /tmp/ctmB.BAAa007EL into MXM.w2f.f, based on
source MXM.f
```

Fragment of the MXM.w2f.f file:

```
C       **** statements ****
C
C       DOACROSS will be converted to SUBROUTINE __mpdo_mxm_1
C$DOACROSS if(((DBLE(__mp_sug_numthreads_func$()) *((DBLE(
C$& __mp_sug_numthreads_func$()) * 1.23D+02) + 2.6D+03)) .LT.((DBLE((
C$& __mp_sug_numthreads_func$() + -1)) *(DBLE(L) * DBLE(N))) * 3.0D00))
C$& ), local(K, I), shared(N, L, C)
        DO K = 1, N, 1
          DO I = 1, L, 1
            C(I, K) = 0.0D00
          END DO
        END DO
C       DOACROSS will be converted to SUBROUTINE __mpdo_mxm_2
C$DOACROSS local(K0, do_ivar, I0), shared(C, A, B, N, L, M)
        DO K0 = 1, N, 1
          DO do_ivar = 0, (INTRN_I4DIVCEIL(M, 4) + -1), 1
            DO I0 = 1, L, 1
              C(I0, K0) = ((((C(I0, K0) +(A(I0, (do_ivar * 4) + 1) * B((do_ivar * 4) + 1, K0)))
+(A(I0, (do_ivar * 4) + 2) * B((do_ivar * 4) + 2, K0))) +(A(I0, (do_ivar * 4) + 3) * B((do_ivar * 4)
+ 3, K0))) +(A(I0, (do_ivar * 4) + 4) * B((do_ivar * 4) + 4, K0)))
            END DO
          END DO
        END DO
```

Usually the –mplist file is more readable than the one generated with –pfa –FLIST:=on

PFA and other optimisations

- Previously there was a delicate relationship between the compiler optimisation level (e.g. –O2) and the preprocessor settings
- This is no longer true, but it is still recommended to use the highest optimisation level
- This is because it allows LNO to do many powerful transformations needed to generate efficient parallel code

Manual Parallelisation

MP Directives and Pragmas

Special FORTRAN comments or C pragmas can be inserted in application sources directing the compiler to generate appropriate parallel code

Features:
- Powerful and easy to use.
- Enabled by compiler option –mp.
- Code can still be maintained for portability and serial execution.
- Can be mixed with message–passing to create hybrid shared/distributed memory programs.

What happens at compile time:
- Parallel region is converted to a subroutine/function.
- Shared variables are passed as arguments.
- Local/private variables are made local to subroutine/function.

Manual Parallelisation – Example

```
SUBROUTINE MXM (A, B, C, L, M, N, N1)
DIMENSION A(M,L), B(N1,M), C(N1,L)

      DO 100 K = 1, L
        DO 100 I = 1, N
          C(I,K) = 0.
100   CONTINUE

c$doacross local(i,j,k), share(a,b,c, L,M,N)
      DO 110 K = 1, L
        DO 110 I = 1, N
          DO 110 J = 1, M
            C(I,K) = C(I,K) + B(I,J) * A(J,K)
110   CONTINUE

      RETURN
      END
```

the only change made

Note: This nested loop will be parallelised for you by the compiler

Manual Parallelisation – Example

```
#pragma parallel
#pragma shared (a,b,c) local(i) byvalue(n)
{
#pragma pfor iterate(i=0;n;1) if (n>100)
   for (i=0; i<n; i++)
       a[i] = b[i] + c[i];

}
```


PFA – Directives and Assertions

- Often the special options need be applied to individual loops
- This is where directives/assertions are used for

  directives – change default settings
  assertions – supply additional information

- Use them on time consuming loops only ...
- Scope of options/directives/assertions:

  Options:
  The entire file

  Directives/assertions:
  The loop that immediately follows it, or from where it appears until end of file

Note:
You may find that the same functionality is covered using different directives/assertions ....

PFA/PCA – Directives and Assertions

Who does what?

- Fortran/C MP directives/pragmas are interpreted *after PFA/PCA* has performed its transformations
- Fortran/C MP directives/pragmas are *not touched by PFA/PCA*
- This means that a source can contain:
  - MP directives/pragmas and/or
  - PFA/PCA directives/pragmas and/or
  - PFA assertions
- This source might then be compiled with special options ....
- Due to all of this, sometimes unwanted interaction can take place
- **Message:** Be careful and check the behaviour

The DOACROSS Directive

DOACROSS – Syntax

In all it's simplicity, the DOACROSS is an extremely powerful directive for shared memory parallelization:

```
c$doacross
c$&    share   (var1, var2, var3, ...),
c$&    local   (var4, var5, var6, ...),
c$&    lastlocal     (var7, var8, var9, ...),
c$&    reduction   (vara, varb, varc, ...),
c$&    if (logical expression)
c$&    mp_schedtype = type,
c$&    chunk       = chunksize
```

- Modifiers are comma separated; can be combined or separated into different lines
- c$& is used for line extensions
- c$ is a comment line unless compiled with –mp
- This can be used to insert code (or calls) to be ignored in serial compilation

The DOACROSS modifiers

share
List of variables that are common between all threads.

local
Variables that are created/destroyed upon entering/leaving a parallel region. Each thread has its own private copy of a local variable.

reduction
Scalar or vector variables used in reduction operations (i.e. sum, dot product, min, max). The compiler generates local variables for partial reduction, and combines locals into reduction variable.

lastlocal (performance penalty to be paid)
Save the value on the last iteration of a local scalar variable.

if (*logical expression*)
Conditionally execute this loop in parallel.

mp_schedtype/chunk
Controls the scheduling of the iterations of the loop.

The DOACROSS

- Nesting of c$doacross is not allowed
- A c$doacross only:
  - The compiler generates a list of localized variables
  - All other data is shared
- I would certainly recommend to include all variables in the share/local list yourself

Why the need to use a DOACROSS?

Consider the following example:

```
 do j=1,5
   do i=1,imax   ! suppose imax = 1000
      b(i,j) = a(i,j)
   end do
end do
```

(higher parallel level)

```
c$doacross local(j,i), share(a,b,imax)
     do j=1,5
        do i=1,imax
           b(i,j) = a(i,j)
        end do
     end do
```

or

(better load balancing)

```
     do j=1,5
c$doacross local(i), share (j,imax,a,b)
        do i=1,imax
           b(i,j) = a(i,j)
        end do
     end do
```

DOACROSS – Lastlocal

```
do j=1,jmax
   do i=1,imax
      x = a(i,j) + b(i,j)
      y = pi2*x
      c(i,j) = y
   end do
end do
write (*,*) 'y=',y
```

the last value of y is needed

avoids lastlocal(y)

```
c$doacross local(i,j,x)
c$&        share(pi2,jmax,imax,a,b,c)
      do j=1,jmax
         do i=1,imax
            x = a(i,j) + b(i,j)
            y = pi2*x
            c(i,j) = y
         end do
      end do
      y = pi2*(a(imax,jmax)+b(imax,jmax))
c or  y = c(imax,jmax)
      write (*,*) 'y=',y
```


DOACROSS – Lastlocal and reduction

```
      sum   = 0.0
      max_a = a(1)
c$doacross lastlocal(i),
c$&        reduction(sum, max_a),
c$&        share(a,imax)
      do 1000 i=1,imax
         sum   = sum + a(i)
         max_a = max(max_a, a(i))
1000 continue
      write (*,*) ' last value: i=',i
```

In this example the value of **i** is preserved and both **sum** and **max_a** are calculated in parallel

The compiler generates the extra code required to do this

DOACROSS – Using the IF qualifier

```
c$doacross local(j,i,tmp),
c$&        if (jmax .gt. min)
      do j=1, jmax
         do i=j+1, imax
            tmp = a(i,j)
            a(i,j) = a(j,i)
            a(j,i) = tmp
         end do
      end do
```

*In this example the **IF** is used to ensure that only for a sufficiently large workload, the loop runs in parallel*

DOACROSS – Using the IF qualifier

```
      num = mp_numthreads()
c$doacross local(i), if (num .eq.2)
      do i=1,2
         if (i.eq.1) call work1
         if (i.eq.2) call work2
      end do

c$doacross if (jmax.gt.imax), local(i)
      do i=1, imax
         e(i) = d(i) + e(i + jmax)
      end do
```

In these examples the **IF** is used to:

- Parallelise the calls to **work1** and **work2**
  (there is a much more elegant way to do this: use the PCF directives, more on this later)
- Guard against a **data dependency** (to ensure right results)

Other Directives and Assertions

With PFA your mileage may vary ....

PFA – Directives and Assertions / 1

```
C*$* CONCURRENTIZE
CVD$ CONCUR
C*$* ASSERT DO (CONCURRENT)
```

- *Don't inhibit parallelism in the loop because of array dependences*
- *Similar to IVDEP, but can be applied to outer, as well as inner loops*

```
C*$* NO CONCURRENTIZE
C*$* ASSERT DO (SERIAL)
CDIR$ NEXT SCALAR
```

(scope is the entire SUBROUTINE!!)

- *Don't parallelise the loop following the directive*

```
C*$* ASSERT PERMUTATION (array1 , array2, .. )
```

With PFA your mileage may vary ....

- *The mentioned arrays are permutation arrays*

PFA – Directives Example/1

```
      subroutine dd(nsite)
      parameter(nx=300,ny=300,ntot=nx*ny,idm=6*ntot)
      common /fake/i1,sxx,tx
      dimension i1(idm/2),sxx(ntot),tx(6)

      do 20 k=1,nsite
         sxx(i1(k))=sxx(i1(k))+force*tx(n)*tx(n)
 20   continue

      return
      end
```

Compiled with –pfa –O3 , the loop will not be parallelised

Why?

PFA – Directives Example/2

```
      subroutine dd(nsite)
      parameter(nx=300,ny=300,ntot=nx*ny,idm=6*ntot)
      common /fake/i1,sxx,tx
      dimension i1(idm/2),sxx(ntot),tx(6)
C*$* ASSERT PERMUTATION (i1)
      do 20 k=1,nsite
         sxx(i1(k))=sxx(i1(k))+force*tx(n)*tx(n)
 20   continue

      return
      end
```

Now the loop will be parallelised

PFA – Directives Example/3

```
ruud@itv> cat dd.l
Parallelization Log for Subprogram dd_

  6: Not Parallel
  Array dependence from sxx on line 7 to sxx on line 7.
  Array dependence from sxx on line 7 to sxx on line 7.


ruud@itv> cat no_dd.l
Parallelization Log for Subprogram no_dd_

    7: PARALLEL (Auto) __mpdo_no_dd_1
```

With PFA your mileage may vary ....

Data Dependencies

- Another assertion that can be used to deal with parallelization of apparent (i.e. "not really, but they look like it") data dependencies:

```
c*$* assert do (concurrent)
   DO i = 1, N
      A(indx(i)) = A(indx(i)) + B(i)
   END DO
```

- Be aware that you are responsible now!
- An obvious data dependency will still not be parallelized *

```
c*$* assert do (concurrent)
   DO i = 1, N
      A(i) = A(i-1) + B(i)
   END DO
```

PFA will NOT parallelize this loop, despite the assertion !!

*) So obvious, you would almost start to feel stupid ...

Assertions: c*$* assert do (serial)

Be aware of the scope of the assertion:

```
DO ....
  DO ....
C*$* ASSERT DO (SERIAL)
      DO ....
        DO ....

        END DO
      END DO

    END DO
END DO
```

PFA will *not* parallelize these outerloops

PFA *may* parallelize one of these loops

Please note that PFA can still perform loop transformations

PFA – Directives and Assertions / 2

`C*$* ASSERT DO PREFER(CONCURRENT)`

- *Specify which loop should be parallelised*

`C*$* ASSERT DO PREFER (SERIAL)`

- *Don't parallelise the loop following the directive,*

Example:

```
c*$* assert do prefer (serial)
        do i=1,L
          do j=1,m
c*$* assert do prefer (concurrent)
              do k=1,n
```

PFA – Directives and Assertions / 3

```
C*$* ASSERT CONCURRENT CALL
CVD$ CNCALL
```

- *Any calls in the following loop (and more nested loops) are ignored when computing cross iteration dependences*
- *This will affect LNO and SWP*
- *In order for this to be safe:*

1. The call can not read data, written in another iteration
2. Can not write data accessed in another iteration
3. Can not read data that will be written somewhere else in multiple iterations (i.e. it can not read data that will otherwise be made thread local)

PFA – Concurrent Call Example

- A procedure call might contain a *data dependency*
- Without prior knowledge, such a call will not be parallelized:

```
DO i = 1, N
  CALL DO_PARALLEL_WORK(i)
END DO
```

- An assertion can be used to cause the loop to be parallelized:

```
c*$* assert concurrent call
   DO i = 1, N
      CALL DO_PARALLEL_WORK(i)
   END DO
```


PFA – Directives and Assertions / 4

`C*$* OPTIMIZE` *(level)*

- *Level is 0, 1, 2 or 3*
- *If level is set to zero, don't do any parallelisation*
- *Otherwise, ignore this directive*

`C*$* ROUNDOFF` *(level)*

- *Level is 0, 1, 2 or 3*
- *If set to zero or one, set –OPT:ro=1 for the duration of LNO*
- *This will affect parallelisation and other LNO optimisations, but it will not affect other phases of the compiler (like SWP)*

Parallelisation Strategy

How to parallelise a code with PFA:

1. Perform profiling to find the time–consuming parts
2. Use PFA/PCA with default settings to auto–parallelise those parts
3. Profile the parallelised code
4. Study the individual behaviour of the parallelised parts
5. If the speed–up is not sufficient, check why
6. Try to find options/directives/assertions to achieve your goal
7. If found, go to step 3
8. If not found, implement changes in the code and go to step 2

Examples of Manual Parallelisation

Example: Loop Peeling

```
ileft = imax
do i=1,imax
   e(i) = (d(i) -d(ileft))/2.0
   ileft =  i
end do
```

Original source

Loop Carried Dependency

Equivalent code

```
      e(1)=(e(1) - d(imax))/2.0
c$doacross local(i), share(e,d)
      do i=2,imax
         e(i) = (d(i) -d(i-1))/2.0
      end do
```

Modified source

no mistake

Example: Inner–loop Strip Mining

```
do 21 k=1,nnn     ! nnn=1000
   do 21 j=1,nnn
      do 21 i=1,nnn
         A(i,j,k)=A(i,j-1,k-1)+A(i,j,k)
21    continue
```

Original source

Observations:

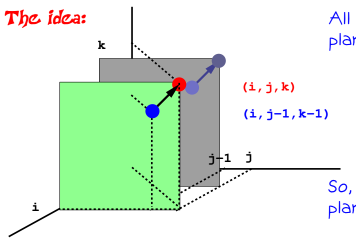
- *The loops are* ***nested properly*** *for memory access*
- *The inner most loop is the* ***only parallelizable loop***
- *The MP* ***overhead*** *would be huge if we parallelize the loop over i*
- ***Memory access time*** *will be increased if the i loop is moved to the outermost*
- ***Solution:*** *use inner–loop strip mining (see next page)*

Example: Inner–loop Strip Mining

```
do 21 k=1,nnn     ! nnn=1000
   do 21 j=1,nnn
      do 21 i=1,nnn
         A(i,j,k)=A(i,j-1,k-1)+A(i,j,k)
21    continue
```

Original source

The idea:



All operations between "i" planes are independent

So, we can process 2D planes in parallel

Example: Inner–loop Strip Mining

Parallel source

```
      nnn        = 1000
      numthreads = mp_numthreads()
      npiece     = nnn/numthreads
      nextra     = mod(nnn, numthreads)
c$doacross local(mmm,is,ie,i,j,k)
      do 100 mmm=1,numthreads

         is = (mmm-1)*npiece + 1
         ie = is + npiece - 1
         if (nextra .ne. 0) then
            is = is + min(nextra,mmm-1)
            ie = ie + min(nextra,mmm)
         end if

         do 21 k=1,nnn
            do 21 j=1,nnn
               do 21 i=1,nnn
c-sgi
                  do 21 i = is, ie
                     a(i,j,k)=a(i,j-1,k-1)+b(i,j,k)
21       continue

100   continue
```


Parallelising code with shared data

In general, shared data needs care when manually parallelising an application

We will study an example that can be typical of a real-life situation

Example: COMMON blocks in Fortran

Parallelising code with shared data

Original source

```
      program main
      parameter (im=1000,jm=1000)
      common/com/a(im,jm),b(im,jm),
     &            c(im,jm),d(im*jm),f(im)

       do j=1,jm
          call do_work(j)
       end do
       stop
       end
       subroutine do_work(j)
       parameter (im=1000, jm=1000)
       common/com/a(im,jm),b(im,jm),
     &            c(im,jm),d(im*jm),f(im)
       const = 3.1415926*j
       do i=1, im
          f(i)   = (a(i,j) + b(i,j))*const
          c(i,j) = f(i)**3
       end do
       return
       end
```

We want to parallelise this loop !

Parallelising code with shared data

Wrong source

```
      program main
      parameter (im=1000,jm=1000)
      common/com/a(im,jm),b(im,jm),
     &            c(im,jm),d(im*jm),f(im)
C$DOACROSS LOCAL(J) SHARED(JM)
       do j=1,jm
          call do_work(j)
       end do
       stop
       end
       subroutine do_work(j)
       parameter (im=1000, jm=1000)
       common/com/a(im,jm),b(im,jm),
     &            c(im,jm),d(im*jm),f(im)
       const = 3.1415926*j
       do i=1, im
          f(i)   = (a(i,j) + b(i,j))*const
          c(i,j) = f(i)**3
       end do
       return
       end
```

WRONG ANSWER GUARANTEED

Parallelising code with shared data

Wrong source

```
   program main
   parameter (im=1000,jm=1000)
   common/com/a(im,jm),b(im,jm),
 &            c(im,jm),d(im*jm),f(im)
C$DOACROSS LOCAL(J) SHARED(JM)
    do j=1,jm
      call do_work(j)
    end do
    stop
    end
    subroutine do_work(j)
    parameter (im=1000, jm=1000)
    common/com/a(im,jm),b(im,jm),
 &            c(im,jm),d(im*jm),f(im)
    const = 3.1415926*j
    do i=1, im
       f(i)    = (a(i,j) + b(i,j))*const
       c(i,j) = f(i)**3
    end do
    return
    end
```

Every thread will write into F() at "random" moments

Parallelising code with shared data

There are two ways to solve this problem:

- *First method (generally applicable)*
  - Put the "local" data in a seperate common block, say this is called common /scratch/
  - Compile with the additional flag:

    –Wl,–Xlocal,scratch_ *the _ is no mistake*
  - This will create a "thread private common block"

  Use c$copyin to copy original common block data from the master thread (if required)

  For example: `C$COPYIN x,y, /foo/, a(i)`
- *Second method (example specific)*
  - *Extend* the array: F(i) is expanded into F(i,"thread_id")

Parallelising code with shared data

Fixed source

```
   program main
   parameter (im=1000,jm=1000)
   common/com/a(im,jm),b(im,jm),
 &            c(im,jm),d(im*jm)
   common /scratch/f(im)
C$DOACROSS LOCAL(J) SHARED(JM)
    do j=1,jm
      call do_work(j)
    end do
    stop
    end
    subroutine do_work(j)
    parameter (im=1000, jm=1000)
    common/com/a(im,jm),b(im,jm),
 &            c(im,jm),d(im*jm)
    common /scratch/f(im)
    const = 3.1415926*j
    do i=1, im
       f(i)    = (a(i,j) + b(i,j))*const
       c(i,j) = f(i)**3
    end do
    return
    end
```

Step #1:
Split the common block

Parallelising code with shared data

Step 2: compile/link as follows:

```
% f77 -O3 -pfa keep -Wl,-Xlocal,scratch_ source.f
```

- This will create a thread private copy of common block "scratch"
- Now every thread can access this common block exclusively

Parallelising code with shared data

Fixed source, fix #2

```
      program main
      parameter (im=1000,jm=1000)
      common/com/a(im,jm),b(im,jm),
     &            c(im,jm),d(im*jm),
     &            f(im,PMAX)
C$DOACROSS LOCAL(J) SHARED(JM)
       do j=1,jm
         call do_work(j)
       end do
       stop
       end
       subroutine do_work(j)
       parameter (im=1000, jm=1000)
       common/com/a(im,jm),b(im,jm),
     &            c(im,jm),d(im*jm),
     &            f(im,PMAX)
       const = 3.1415926*j
       do i=1, im
          f(i,"TID")=(a(i,j)+b(i,j))*const
          c(i,j) = f(i)**3
       end do
       return
       end
```

Make F a 2D array

"TID" is the Thread ID

no mistake

Parallelising code with shared data

The details:

- The "TID" can be obtained from the MP function:

  mp_my_threadnum()

- Like this:

```
TID = mp_my_threadnum() + 1
```

no mistake

no mistake

About Variables

Global variables in a serial program

Variables that can be accessed in multiple procedures:

- Arguments to functions and subroutines
- Variables defined in common blocks

Essentially it means that *the same memory location* can be accessed from different places in the code

About Variables

Local variables in a serial program

Variables that can be accessed only within the procedure that they have been defined in:

- Automatic variables
  Automatically created at each time the procedure is called. They are destroyed upon exit of the procedure.
- Static variables
  Statically allocated at compiling time. They are initialized to zero, and exist for the life of the program. When a procedure is compiled with option –static, all its local variable are allocated as static variables. Variables defined through save and data statements are also static variables.
- Dynamic variables
  Variables defined through malloc

About Variables

Variables in a parallel program

- Shared serial variables
  Can be accessed by every thread. Independent read/write operations can take place.
- Local serial variables
  Every thread has it's own copy of the variables that are created/destroyed upon entering/leaving the procedure. They are not visible to other threads.

Serial Code	Parallel Code
global variables	shared variables
auto local variables	local variables
static variables	*not allowed*
dynamic variables	use with care

So, we are done ...

Parallelisation of Procedure Calls

Check the procedure for:

- Scratch variables (can use automatic variables only)
  - Do not compile with the –static option
  - Do not pass *scratch* variables through a common block
  - Do not use save or data statements
- Usage of common blocks (use them with care)
  - Only truly shared variables in common block
    no action needed
  - Only local variables in the common block
    compile –Wl,–Xlocal,*commonblock*_
  - A *mixture* of shared and local variables
    split the common block and compile –Wl,–Xlocal,....

Parallelising an Inner Product

```
      implicit double precision (a-h, o-z)
      parameter (imax = 1000000, kmax=4)
      dimension sum(0:kmax-1),a(imax,0:kmax-1)

       < initialize sum and a >

c$doacross local(i,k), share(a, sum)
       do k=0,kmax-1
          do i=1,imax
             sum(k) = sum(k) + a(i,k)
          end do
       end do

       stop
       end
```

Original source

This will run slow

Why?

Parallelising an Inner Product

```
      implicit double precision (a-h, o-z)
      parameter (imax = 1000000, kmax=4)
      dimension sum(0:kmax-1),a(imax,0:kmax-1)

       < initialize sum and a >

c$doacross local(i,k), share(a, sum)
       do k=0,kmax-1
          do i=1,imax
             sum(k) = sum(k) + a(i,k)
          end do
       end do

       stop
       end
```

Array SUM is shared 4 Million times!

Original source

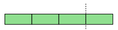
SUM(0)	SUM(1)	SUM(2)	SUM(3)
CPU 1	CPU 2	CPU 3	CPU 4

Parallelising an Inner Product

This is an example of false sharing of cache lines:

```
c$doacross local(i,k), share(a, sum)
 do k=0,kmax-1
    do i=1,imax
       sum(k) = sum(k) + a(i,k)
    end do
 end do
```

- Thread "k" will read and write sum(k)



- However, elements sum(0) ... sum(3) are within 1 (or 2) cache lines

Two different solutions:

1. Create thread private variables and merge them later
2. Create an array sum(16,0:kmax–1) and waste 15 elements per thread by using sum(1,k) only.
   All these values are then one cache line apart and don't share modified data elements

Parallelising an Inner Product

```
      implicit double precision (a-h, o-z)
      parameter (imax = 1000000, kmax=4)
      dimension sum(0:kmax-1),a(imax,0:kmax-1)

       < initialize sum and aa >

c$doacross local(i,k,ss), share(a, sum)
       do k=0,kmax-1
          ss = 0.0
          do i=1,imax
             ss = ss + a(i,k)
          end do
          sum(k) = ss
       end do
```

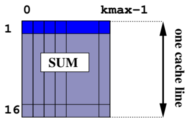
Modified source
Solution #1

- *Previously, SUM was shared 4 million times*
- *Now SUM is only shared 4 times*
- *On P processors, only P additional elements needed*
- *This version scales very efficiently*

Parallelising an Inner Product

```
      implicit double precision (a-h, o-z)
      parameter (imax = 1000000, kmax=4)
      dimension
     sum(16,0:kmax-1),a(imax,0:kmax-1)

       < initialize sum and aa >

c$doacross local(i,k), share(a, sum)
       do k=0,kmax-1
          do i=1,imax
             sum(1,k) = sum(1,k) + a(i,k)
          end do
       end do
```

Modified source
Solution #2



False Sharing of Cache Lines

If your program doesn't scale as expected

BE ON THE LOOKOUT FOR A FALSE SHARING PROBLEM

Detecting False Sharing:

- Not an issue for read–only and write–only data
- Be alert when:
  - Data is read and written
  - Data is accessed in a one dimensional way

Parallel Computing Forum (PCF) directives

An Example of PCF directives

```
C$PAR PARALLEL local(i) shared(a,b,c)
C$PAR PDO
      do i=1,N
         a(i) = b(i) + c(i)
      enddo
C$PAR END PDO
C$PAR END PARALLEL
```

Optional

Activate PCF directives by compiling (and linking) with:

–pfa [keep] or
–mp

The PCF directives – Overview

C$PAR&
Continuation line

C$PAR PARALLEL DO
Equivalent of the "doacross"

C$PAR [END] CRITICAL SECTION
All processors execute this part, but one processor at a time gets access

C$PAR [END] PARALLEL
Encloses a parallel region

- C$PAR BARRIER
  Explicit synchronization
- C$PAR [END] PDO
  Parallel execution of a loop
- C$PAR [END] SINGLE PROCESS
  Only one processor executes this part
- C$PAR [END] PSECTION [S] –– Encloses a parallel section
  - C$PAR SECTION
    Defines indivual blocks of code, each block executed in parallel

The PCF directives – Syntax

C$PAR PARALLEL DO [*clause* [[,] *clause*] ...]

The clauses are the same as with DOACROSS:

= NOT the preferred usage !

[IF (logical_expression)]
[{LOCAL | PRIVATE} (item[,item ...])]
[{SHARE | SHARED} (item[,item ...])]
[{LASTLOCAL | LAST LOCAL} (item[,item ...])]
[REDUCTION (item[,item ...])] *(+,*, min(), max())*
[MP_SCHEDTYPE=mode]
[CHUNK=integer_expression]

```
C$PAR PARALLEL DO LOCAL(i) SHARE(a,b,c)
      do i=1,N
         a(i) = b(i) + c(i)
      end do
```

The PCF directives – Syntax

C$PAR CRITICAL SECTION [*(lock_variable)*]
"code"
C$PAR END CRITICAL SECTION

- This construct can appear ANYWHERE within a program
- The critical section is executed by one processor at a time
- The optional integer lock_variable has to be initialized to zero

```
C$PAR CRITICAL SECTION
      sum = sum + a(i)
C$PAR END CRITICAL SECTION
```


The PCF directives – Syntax

C$PAR PARALLEL [*clause* [[,] *clause*] ...]
"code"
C$PAR END PARALLEL

The clauses are:

[IF (logical_expression)]
[{LOCAL | PRIVATE} (item[,item ...])]
[{SHARED | SHARE} (item[,item ...])]

Note that here SHARED is the preferred notation!

All processors execute the parallel region

```
C$PAR PARALLEL LOCAL(i)
      i = mp_my_threadnum()
      call do_work(i)
C$PAR END PARALLEL
```


The PCF directives – Syntax

C$PAR BARRIER

- This will cause <u>all processors</u> to wait at the specified point
- Only when all of them have reached the point, execution continues
- This is an expensive operation and should be used with care

```
      do i=1,N
         a(i) = b(i) + c(i)
      end do
C$PAR BARRIER
```


The PCF directives – Syntax

C$PAR PDO [*clause* [[,] *clause*] ...]
"code"
[C$PAR END PDO [NOWAIT]]

- **Clauses:**
  [{LOCAL | PRIVATE} (item[,item ...])]
  [{LASTLOCAL | LAST LOCAL} (item[,item ...])]
  [(ORDERED)]
  [sched]
  [chunk]
- **Notes:**
  - A SHARED item may be declared as LOCAL !
  - ORDERED ~ DYNAMIC & CHUNK=1
  - sched and chunk are as in the DOACROSS
  - NOWAIT : proceed to the code after the directive

The PCF directives – Syntax

C$PAR SINGLE PROCESS [*clause* [[,] *clause*] ...]
"code"
[C$PAR END SINGLE PROCESS [NOWAIT]]

- **Clauses:**
  [{LOCAL | PRIVATE} (item[,item ...])]
- **Notes:**
  - **A SHARED item may be declared as LOCAL !**
  - **NOWAIT : proceed to the code after the directive**

This directive will have the effect that only one (and one only!) thread will execute the code enclosed

Meanwhile all other threads are blocked!

The PCF directives – Syntax

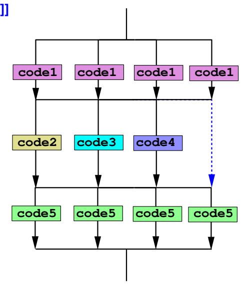
C$PAR PSECTION[S] [*clause* [[,] *clause*] ...]
"code"
[C$PAR SECTION
"code"]
C$PAR END PSECTION[S] [NOWAIT]]

- **Clauses:**
  [{LOCAL | PRIVATE} (item[,item ...])]
- **Notes:**
  - **A SHARED item may be declared as LOCAL !**
  - **NOWAIT : proceed to the code after the directive**
  - **Nested parallelism is not possible:**
    A section can not contain DOACROSS, PDO, PARALLEL DO, or SINGLE PROCESS

The PCF directives – Syntax

C$PAR PSECTION[S] [*clause* [[,] *clause*] ...]
"code"
[C$PAR SECTION
"code"]
C$PAR END PSECTION[S] [NOWAIT]]

```
C$PAR PARALLEL

      "code1"

C$PAR       PSECTIONS
                  "code2"
C$PAR          SECTION
                  "code 3"
C$PAR          SECTION
                  "code 4"
C$END       PSECTION

      "code 5"

C$PAR END PARALLEL
```



The PCF directives – Summary

- ***These directives allow for very flexible parallel programming***
- ***Due to their nature, they contain more synchronization points than the simple DOACROSS***
- ***Therefore, avoid them when a DOACROSS can do the same***
- ***Use the (PCF) directives at the highest parallel level possible***

Manual Parallelisation in C

Parallel Regions in C

- Manual parallelization in C makes use of #pragma
- These are generated using the –pfa option
- Manually inserted pragmas are recognized using –mp
- The syntax of pragmas is very similar to the *PCF directives**

There is one **big** difference:

ALL parallel constructs are enclosed in a parallel region

*) because they're modelled after them ....

Parallel Regions in C

All parallel regions in a C program are enclosed by a

#pragma parallel

```
#pragma parallel
{
 /* Whatever you want to run in parallel */
}
```

- The code contained *within* the parallel region is executed by all the threads
- A parallel region can contain other parallelization pragma's

Parallel Regions in C

Parallel C pragmas are flexible in lay–out:

```
#pragma parallel shared(a,b,c,n) local(i) pfor
#pragma iterate(i=0;n;1)
        for(i=0;i<n;i++)
             a[i] = b[i] + c[i];
```

This is equivalent to:

```
#pragma parallel
#pragma shared (a,b)
#pragma shared (c,n)
#pragma local(i)
#pragma pfor
#pragma iterate (i=0;n;1)
        for(i=0;i<n;i++)
             a[i] = b[i] + c[i];
```

Parallel Regions in C

General rules for #pragma:

- Syntax: #pragma keyword [modifier(s)]
- Separate keywords and modifiers by spaces
- Scope of a pragma:
  - The succeeding statement
  - A compound statement enclosed by {}
- The #pragma parallel can not be nested
- Pragma parallel and pfor can be continued on other lines
  (don't repeat the keywords!)
- Parallelized loops can perform I/O

Parallel Pragmas Overview

#pragma parallel [{}]
Encloses a parallel region

#pragma pfor [{}]
Distribute the iterations of a loop over the threads

#pragma one processor [{}]
Only one thread will execute this part

#pragma critical [{}]
All threads will execute this part, but only one at a time

#pragma synchronize
Execution continues after all threads have reached this point

#pragma enter/exit gate (*)
No thread exits until they all have entered the region

#pragma independent [{}] (*)
Execute arbitrary (unrelated) regions in parallel

*) Not generated by PFA yet, but you can insert them by hand

Pragma parallel

```
#pragma parallel
#pragma shared(variables) local(variables)
#pragma [modifiers]
{
 Encloses this parallel region
}
```

Optional modifiers:

if (integer valued expr)	don't run parallel if expression equals zero (false)
numthreads (expr)	use "expr" threads
numthreads (percent=expr)	use expr% of the available threads
numthreads(min=expr;max=expr)	min: don't run unless >= min threads available max: never run on more than max threads
byvalue (variables)	pass variables by value instead of by reference
noifinline	fine-tuning in conjunction with "if"

Pragma pfor

```
#pragma pfor iterate(index=expr1; expr2; expr3)
#pragma [modifiers]
{
 execute iterations of this for (......) loop in parallel
}
```

The parameters for iterate:

index	loop variable
expr1	starting value of loop variable
expr2	number of iterations performed
expr3	increment of the loop variable

Optional modifiers:

schedtype (type)	type defines the scheduling type: simple, interleave, gss, dynamic, runtime
chunksize (expr)	chunksize for interleave or dynamic (gss: chunk ~ iterations_left/(2*P)+1)

Pragma pfor – Examples

Example of a parallelised loop

```
for (i=0; i<n; i++)
       a[i] = b[i] + c[i];
```

```
#pragma parallel
#pragma shared (a,b,c) local(i) byvalue(n)
{
#pragma pfor iterate(i=0;n;1) if (n>100)
   for (i=0; i<n; i++)
       a[i] = b[i] + c[i];

}
```


Pragma pfor – Examples

```
#pragma parallel local(...) shared(...)
{
  #pragma pfor iterate (i=0; IMAX; 1)
  #pragma schedtype(interleave) chunksize(2)
   for (i=0; i<IMAX; i++)
   {
       /* do some work  */
   }
}
```

Optional

Proc 1:	0	1	8	9	16	17
Proc 2:	2	3	10	11	18	19
Proc 3:	4	5	12	13	20	21
Proc 4:	6	7	14	15	22	23

iterations

Pragma pfor – Examples

```
pi2 = 2.0*3.14159;
#pragma parallel local(i,j) shared(a,b)
#pragma byvalue(imax, jmax, pi2)
{
    #pragma pfor iterate (i=0; imax; 1)
    for (i=0; i<imax; i++)
    {
       for (j=0; j<jmax; j++)
       {
          b[i][j] = pi2*a[i][j];
       }
    }
}
```


parallel&pfor – doacross

PARALLEL/PFOR is similar to the DOACROSS, but does not:

- **Have default variable types**
- **Support *reduction* and *lastlocal***

PARALLEL/PFOR does support:

- **Byvalue: read-only scalar variables (slightly more efficient)**
- **Flexible control over the number of threads**
  #pragma numthreads (N) / (percent=*expr*) / (min=*expr*; max=*expr*)
  (Fortran uses the simpler *mp_set_numthreads(N)* utility subroutine)

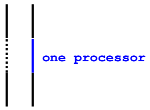
Note:
If both pragma and environment variables are used to set the number of threads, the <u>smaller</u> value of them will be used.

Pragma one processor

#pragma one processor
{
only one thread executes: **code**
}

Note that all other threads are blocked meanwhile

```
#pragma parallel
{
   do_parallel_work();

#pragma one processor
  {
       assemble_results();
  }

   prepare_next_step();
}
```

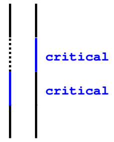


Pragma critical

#pragma critical [lock(locktype)]
{
sequentially all threads execute: **code**
}

Note that always P–1 threads are waiting for access to the critical region

```
#pragma parallel
{
   do_parallel_work();

#pragma critical
  {
       update_residual();
  }

   prepare_next_step();
}
```



About the lock

lock (locktype)

This modifiers lets you control what kind of lock you want to use during the execution of a parallel code segment.

Choices available for locktype:

- **block**
  Use a lock exclusively for the block representing the parallel code segment
- **region**
  Use a lock that is unique to the parallel region
- **global**
  Use a lock that is unique to the parallel runtimes
- **"your_lock_name"**
  Use a lock that must be provided by you. The name should correspond to a user–defined variable. It is your responsibility to acquire and dispose of the lock (man ussetlock, man usunsetlock)

Defaults:

critical segment	locktype=global
other segments	locktype=block

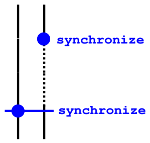
Pragma synchronize

#pragma synchronize

wait untill all threads have reached this point

Note that this is expensive and serializes the computations

```
#pragma parallel
{
   do_parallel_work();

#pragma synchronize

   prepare_next_step();
}
```



Pragma synchronize

- Outside of PCF, this pragma does not have an equivalent Fortran directive
- The same functionality can be realized with an MP function:

mp_barrier()

```
#pragma parallel
{
#pragma pfor (...)
 for (....)
 {
   do_some_work();

   #pragma synchronize
   do_more_work();
 }
}
```

C

```
c$doacross
    do ...
      call do_some_work()

      call mp_barrier()
      call do_more_work()
    end do
```

Fortran

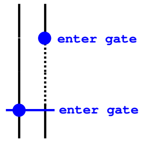
Pragma gate

#pragma enter gate (name)
no thread exits until they all have reached this point
#pragma exit gate (name)

Note that this is an expensive operation

NOT YET GENERATED BY PCA

```
#pragma parallel
{
   do_parallel_work();

#pragma enter gate (gate_1)

   prepare_next_step();

#pragma exit gate (gate_1)

   do_some_more_parallel_work();
}
```



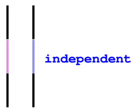
Pragma independent

#pragma independent
{
only one thread executes this block of **code**
}

Possible to use multiple indepedent blocks

NOT YET GENERATED BY PCA

```
#pragma parallel
{
   do_parallel_work();

#pragma independent
    { do_part_A ();}
#pragma independent
    { do_part_B ();}

   exchange_data();
}
```



Pragma independent

- This pragma allows for true MIMD parallel programming
- In Fortran the PCF directives can be used to achieve the same

```
#pragma parallel
{
#pragma independent
   /* task 1 */
#pragma independent
   /* task 2 */
#pragma pfor (...)
    for (....)
}
```

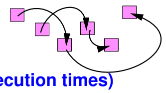
C

```
C$PAR PARALLEL
C$PAR   PSECTIONS
           "task 1"
C$PAR   SECTION
           "task 2"
C$PAR   SECTION
            do ...
            end do
C$END   PSECTION
C$PAR END PARALLEL
```

Fortran

Linked Lists

Linked lists are characterized by:



- Unknown lengths
- Different lengths (i.e. varying execution times)

The PCA pragmas provide help in two cases:

Unordered lists

Loops in which each iteration processes a *different member* of the list and the computations for each element are *independent* of each other

Ordered lists

Loops in which each iteration processes a *different member* of the list and the computations for each element are *to a large extent independent* of each other, but have a small portion of the code which has to be processed in the order in which the elements appear in the list.

Linked Lists

```
#pragma plist unordered (list vars.; initialize shared;
                         initialize local; condition; increment)
for (initialize shared, initialize local; condition; increment)
{
    ... /* loop body */ ....
}
```

```
#pragma parallel shared (llist,head,sum,cnt) local(psum)
{

  #pragma plist unordered (llist; llist=head,sum=0,cnt=0;
                          psum=0;llist; llist = llist->next;)

  for(llist=head,sum=0,cnt=0,psum=0;llist;llist=llist->next)
  {
          /* unordered work to be executed in parallel */
  }

}
```


Linked Lists

```
#pragma plist ordered (list vars.; initialize shared;
                       initialize local; condition; increment)
for (initialize shared, initialize local; condition; increment)
{
    ... /* loop body */ ....
}
```

```
#pragma parallel shared (....) local(....)
{
    #pragma plist ordered (.....)
    for ( .... )
    {
        /* unordered part of for loop  */

        #pragma ordered
        {
            /* ordered part of for loop */
        }

        /* unordered part of for loop  */
    }
}
```


Parallel Pragmas – Summary

#pragma parallel *modifier*

#pragma pfor *modifier*

#pragma one processor

#pragma critical [lock(locktype)]

#pragma independent

#pragma synchronize

#pragma enter gate (*name*)/exit gate(*name*)

#pragma plist

#pragma ordered

Parallel Pragmas – Summary

Parallel code segments within a parallel region

Work–sharing

- *for loop* executed in parallel — #pragma pfor
- *identical code* executed by all threads: — keep it outside of the pfor
- *thread specific code* executed in parallel — #pragma independent

Single threaded execution

- critical sections executed by *one thread at a time* — #pragma critical
- code executed by *only one thread* — #pragma one processor

Explicit synchronization

- *barriers* — #pragma synchronize
- gated *synchronization zones* — #pragma enter/exit gate

The old NAS Kernels (NASA–Ames *)

The results on a PowerChallenge R8K; 1 and 4 processors using –pfa "blindly"

Note the difference in answers

THE NAS KERNEL BENCHMARK PROGRAM

PROGRAM	ERROR	FP OPS	SECONDS	MFLOPS
MXM	3.4313E-15	4.1943E+08	1.8762	223.55
CFFT2D	1.2345E-13	4.9807E+08	5.4087	92.09
CHOLSKY	5.2449E-12	2.2103E+08	5.3650	41.20
BTRIX	2.2163E-13	3.2197E+08	7.7659	41.46
GMTRY	1.8797E-13	2.2650E+08	3.8793	58.39
EMIT	5.9125E-16	2.2605E+08	4.2342	53.39
VPENTA	1.2365E-14	2.5943E+08	3.7571	69.05
TOTAL	5.7943E-12	2.1725E+09	32.2864	67.29

PROGRAM	ERROR	FP OPS	SECONDS	MFLOPS
MXM	3.4313E-15	4.1943E+08	0.4990	840.61
CFFT2D	1.2345E-13	4.9807E+08	6.1284	81.27
CHOLSKY	5.2449E-12	2.2103E+08	15.3120	14.44
BTRIX	2.2163E-13	3.2197E+08	10.1674	31.67
GMTRY	2.7463E-13	2.2650E+08	1.1259	201.17
EMIT	5.9125E-16	2.2605E+08	1.5002	150.68
VPENTA	1.2365E-14	2.5943E+08	1.7592	147.47
TOTAL	5.8810E-12	2.1725E+09	36.4921	59.53

WOW!

*) Don't confuse these with the NAS Parallel Benchmark !

The CHOLSKY Kernel

The results on 1 and 4 processors using –pfa:

```
                PROGRAM     SECONDS      Mflop/s
1 processor:    CHOLSKY      5.3650       41.20
4 processors:   CHOLSKY     15.3120       14.44
```

(3x slowdown!)

What does the profile show?

Parallelisation introduces this

```
1 processor profile:

Each sample covers 4 bytes for every 10.0ms ( 0.18% of 5.4700sec)

samples   time(%)       cum time(%)      procedure (file)
    427   4.3s( 78.1)   4.3s( 78.1)      CHOLSK.PREGION0 (cholsk.f)


Profile of each thread of 2-processor run:

samples   time(%)       cum time(%)      procedure (file)
    800     8s( 68.8)     8s( 68.8)      CHOLSK.PREGION0 (cholsk.f)
    154   1.5s( 13.3)   9.5s( 82.1)      __mp_end_pdo (libmp/mp_pdo.c)

samples   time(%)       cum time(%)      procedure (file)
    836   8.4s( 73.9)   8.4s( 73.9)      CHOLSK.PREGION0 (cholsk.f)
     95  0.95s(  8.4)   9.3s( 82.2)      __mp_end_pdo (libmp/mp_pdo.c)
```

The PFA output for this kernel

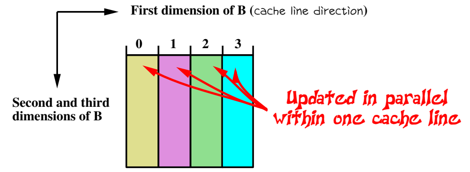
```
                              36   C
     C       +---------       37          DO 6 I = 0, NRHS
            *+--------        38            DO 7 K = 0, N
            *!+-------        39              DO 8 L = 0, NMAT
9     DD SO *!!_______        40   8            B(I,L,K) = B(I,L,K) * A(L,0,K)
10 11 SO    *!+-------        41              DO 7 JJ = 1, MIN (M, N-K)
            *!!+------        42                DO 7 L = 0, NMAT
9 12  DD SO *!!!______        43   7              B(I,L,K+JJ) =B(I,L,K+JJ)-A(L,-JJ,K+JJ)*B(I,L,K)
            *                 44   C
      SO    *+--------        45            DO 6 K = N, 0, -1
            *!+-------        46              DO 9 L = 0, NMAT
13    DD SO *!!_______        47   9            B(I,L,K) = B(I,L,K) * A(L,0,K)
10 11 SO    *!+-------        48              DO 6 JJ = 1, MIN (M, K)
            *!!+------        49                DO 6 L = 0, NMAT
13 14 DD SO *!!!______        50   6            B(I,L,K-JJ) = B(I,L,K-JJ)-A(L,-JJ,K)*B(I,L,K)
                              51   C
                              52         RETURN
                              53         END
   Loop Summary

           From    To      Loop      Loop     at
   Loop#   line    line    label     index    nest    Status

   15      38      51      Do 6      I        2       concurrentized
   16      39      44      Do 7      K        3       already in a parallel loop
   17      40      41      Do 8      L        4       already in a parallel loop
   18      42      44      Do 7      JJ       4       already in a parallel loop
   19      43      44      Do 7      L        5       already in a parallel loop
   22      46      51      Do 6      K        3       already in a parallel loop
   23      47      47      Do 9      L        4       already in a parallel loop
   24      49      51      Do 6      JJ       4       already in a parallel loop
   25      50      51      Do 6      L        5       already in a parallel loop
```

With PFA your mileage may vary....

NRHS=3 and so B(0:NRHS) is in one cache line: another false sharing problem

CHOLSKY: False Sharing

```
DO 6 I = 0, NRHS
   DO 7 K = 0, N
      DO 8 L = 0, NMAT
         B(I,L,K) = B(I,L,K) * A(L,0,K)
```



Additional problem:
NRHS=3 and so parallelism is always restricted to 4 processors

With PFA your mileage may vary....

Tuning CHOLSKY

The original code

```
     DO 6 I = 0, NRHS
        DO 7 K = 0, N
           DO 8 L = 0, NMAT
8             B(I,L,K) = B(I,L,K) * A(L,0,K)
        DO 7 JJ = 1, MIN (M, N-K)
           DO 7 L = 0, NMAT
7             B(I,L,K+JJ) = B(I,L,K+JJ) - A(L,-JJ,K+JJ)*B(I,L,K)

        DO 6 K = N, 0, -1
           DO 9 L = 0, NMAT
9             B(I,L,K) = B(I,L,K) * A(L,0,K)
           DO 6 JJ = 1, MIN (M, K)
              DO 6 L = 0, NMAT
6                B(I,L,K-JJ) = B(I,L,K-JJ) - A(L,-JJ,K) * B(I,L,K)
```

N = 40
NMAT = 250

Loop over L is longest and so has the preference:

- Better software pipelining efficiency (less start/stop overhead)
- Parallelism is not restricted to a relatively low number of processors

Problem: access is not in the last dimension (i.e. not optimal)
Remedy: change dimension declaration of array B

Tuning CHOLSKY

The modified code

```
     DO 6 L = 0, NMAT
        DO 6 I = 0, NRHS
           DO 7 K = 0, N
              B(K,I,L) = B(K,I,L) * A(0,K,L)
           DO 7 JJ = 1, MIN (M, N-K)
7             B(K+JJ,I,L) = B(K+JJ,I,L) - A(-JJ,K+JJ,L)*B(K,I,L)

           DO 6 K = N, 0, -1
              B(K,I,L) = B(K,I,L) * A(0,K,L)
              DO 6 JJ = 1, MIN (M, K)
6                B(K-JJ,I,L) = B(K-JJ,I,L) - A(-JJ,K,L) * B(K,I,L)
```

Loop over L is now the outermost loop

CHOLSKY: Performance Comparison

Original version:

Processors	Seconds	Mflop/s
1	5.37	41.2
4	15.31	14.4
8	this doesn't make any sense to even run	

Modified version:

Processors	Seconds	Mflop/s
1	9.14	24.2
4	2.59	85.4
8	1.51	146.0
16	0.95	232.8

All parallelization is still performed by PFA !

Parallelising Procedure Calls

NAS Kernel *BTRIX (lines deleted for clarity):*

Where does this K come from?

```
      SUBROUTINE BTRIX (JS, JE, LS, LE, K)

      PARAMETER (JD=30, KD=30, LD=30, MD=30)
      COMMON /ARRAYS/ S(JD,KD,LD,5), A(5,5,MD,MD), B(5,5,MD,MD),
     $     C(5,5,MD,MD)

      DO 100 J   = JS,JE

      IF(J.EQ.JS) GO TO 4
       DO 3 M = 1,5
        DO 3 N = 1,5
          DO 3 L = LS,LE
            B(M,N,J,L) = B(M,N,J,L) - A(M,1,J,L)*B(1,N,J-1,L)
     $           - A(M,2,J,L)*B(2,N,J-1,L) - A(M,3,J,L)*B(3,N,J-1,L)
     $           - A(M,4,J,L)*B(4,N,J-1,L) - A(M,5,J,L)*B(5,N,J-1,L)
    3   CONTINUE

  100 CONTINUE

      DO 200 J = JEM1,JS,-1
        DO 200 M = 1,5
          DO 200 L = LS,LE
            S(J,K,L,M) = S(J,K,L,M) - B(M,1,J,L)*S(J+1,K,L,1)
     $           - B(M,2,J,L)*S(J+1,K,L,2) - B(M,3,J,L)*S(J+1,K,L,3)
     $           - B(M,4,J,L)*S(J+1,K,L,4) - B(M,5,J,L)*S(J+1,K,L,5)
  200 CONTINUE

      RETURN
      END
```

Is this really needed?

Array S is updated here

With PFA your mileage may vary ....

Parallelising Procedure Calls

The routine that calls *BTRIX (lines deleted for clarity):*

```
      DO 120 K = 1, KD
         CALL COPY (NB, BX(1,K), B)
         CALL BTRIX (JS, JE, LS, LE, K)
120   CONTINUE
```

- *COPY* initializes B to a column of BX
- Then B is used in *BTRIX* to compute a slice of S (see previous page)
- If B is made *local* instead of *shared* (COMMON), this loop is parallelizable

```
c$doacross local(k,b)
        DO 120 K = 1, KD
         CALL COPY (NB, BX(1,K), B)
         CALL BTRIX (JS, JE, LS, LE, K, B)
120   CONTINUE
```

Added to argument list instead of passed in through a common block

Caveats:

- All local variables of a parallel-called subroutine must be dynamically allocated.
- This means parallel-called subroutine cannot be compiled *–static.*

BTRIX: Performance Comparison

Original version:

Processors	Seconds	Mflop/s
1	7.91	40.7
4	10.71	30.1

Modified version:

Processors	Seconds	Mflop/s
1	9.13	35.3
4	2.58	124.8
8	1.32	243.6
16	0.76	425.1

All parallelization is still performed by PFA !

Portable Shared Memory Programs

- The summation is an example of a reduction operation

  sum = sum + a(i)

- Other reduction operations are:
  - Innerproduct
  - Maximum of a vector
  - Minimum of a vector
- The compiler knows quite well how to parallelize these
- As an example we will show how to parallelize it by hand
- This can be done without violating portability!

With PFA your mileage may vary ....

A parallelised reduction

```
c    This is a potentially parallel loop
        DO 20 I=1,NATOMS

         <calculation>

          IPTR = ...
          ARRAY(IPTR) = ARRAY(IPTR) + X

 20     CONTINUE
```

We could settle for locking the ARRAY updates, if <calculation> is long enough to offset locking overhead:

```
c$      call mp_setlock
          ARRAY(IPTR) = ARRAY(IPTR) + X
c$      call mp_unsetlock
```

Or, we can have a separate array to accumulate contributions for each thread, then combine the threads' contributions into ARRAY

Note that we use c$ to exploit that these calls are only recognized when using -mp

With PFA your mileage may vary ....

A parallelised reduction

```
        DIMENSION ARRAY(MAXARR), AR_LOC(MAXARR,MAXTHR)

c  Initialize ARL_LOC to accumulate contributions.
        NUMTHR = 1
C$      NUMTHR = mp_numthreads()
C$DOACROSS LOCAL(I,ITHR), SHARE(AR_LOC)
        DO 10 ITHR=1,NUMTHR
          DO 10 I=1,MAXARR
 10         AR_LOC(I,ITHR) = 0.0

c   This is a potentially parallel loop
C$DOACROSS LOCAL(I,ITHR,IPTR,X,...),SHARE(AR_LOC)
        DO 20 I=1,NATOMS
           ITHR = 1
C$         ITHR = 1 + mp_my_threadnum()

        <calculation>

           IPTR = ...
           AR_LOC(IPTR,ITHR) = AR_LOC(IPTR,ITHR) + X

 20      CONTINUE

c   Combine contributions into ARRAY

C$DOACROSS LOCAL(I,ITHR), SHARE(ARRAY,AR_LOC)
          DO 30 I=1,MAXARR
            DO 30 ITHR=1,NUMTHR
   30         ARRAY(I) = ARRAY(I) + AR_LOC(I,ITHR)
```


Shared Memory Tuning – Summary

1. Parallelize *outermost* loops possible, *largest iteration counts* possible
2. Avoid *false aliasing* caused by nearby parallel read/writes in shared arrays. Rules of thumb:
   - *Parallelized index in multidimensional arrays should be rightmost (FORTRAN) or leftmost (C)*
   - *Avoid parallel modifications to small 1D vectors*
3. Examine profiles for signs of *load imbalance*, and experiment with *mp_schedtype* to improve load balancing
4. Examine profiles for high parallel overheads, reduce if needed by implementing *parallelism at coarser–grained level*

Shared Memory Parallelisation

Conclusions

- ***Auto–Parallelising compilers can do a lot***
- ***However, they are limited in their capabilities***
- ***This also depends on the application area***
- ***Manual parallelisation is easy to implement***
- ***Don't confuse algorithm and implementation***
- ***For scalable performance one may have to look at the datastructure details***

Appendix