SGI Shared Memory Parallelisation

(An Overview)

Ruud van der Pas

SiliconGraphics

European High Performance Computing Team

MP Programming on SGI systems

The Shared Memory Model

- **Loop Decomposition**
  - Loops in the serial code are parallelised
  - Don't need to worry about where the data is
- **Processors communicate and synchronize via shared memory**
  - Handled by SGI MP library and system calls
- **Easy to program and flexible**
  - Parallelised code is a slightly modified version of the serial code
  - Modifications can be done incrementally, at a pace suitable to the user
  - User does not need to worry about communication between processors: it is implicit in the shared memory model
  - Suitable for both fine– and coarse–grained parallelism, and can also run message–passing code very efficiently

MP Programming on SGI systems

1. Automatic parallelization via PFA:

   Fortran and C compiler option: –pfa

2. Parallelization by inserting SGI compiler directives

   Fortran: c$doacross, c$par, ...
   C: #pragma parallel, ...
   (compile and link code using the –mp option)

3. Low–level parallelisation

   Posix Standard: Pthreads
   SGI library: sproc()
   UNIX native: fork()
   Sequent library: m_fork()

4. Message Passing PVM, MPI
   HPF: pghpf, xHPF (third party))

= Available from SGI

Shared Memory Parallelisation

About shared memory parallelization

- Compilers parallelize at the loop level
- In a nested loop, the highest level is chosen
- It is possible to control the parallelization:
  - From the outside using options and/or
  - From the inside using directives/pragmas
- Manual parallelization allows for any form of parallelization (typically at a high level)

Automatic Parallelisation – Example/1

```
program main
implicit none

integer i, n
parameter (n=1000)
real     a(n), b(n)
```

Parallel

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

Not–Parallel

```
do i = 2, n
   a(i-1) = b(i) + a(i)
end do
do i = 1, n
   print *,'a(',i,')=',a(i),' b(',i,')=',b(i)
end do
```

```
stop
end
```

Automatic Parallelisation – Example/2

Compiler Messages:

```
Parallelization Log for Subprogram MAIN__

  8: PARALLEL (Auto) __mpdo_MAIN__1

13: PARALLEL (Auto) __mpdo_MAIN__2

17: Not Parallel
     Array dependence from a on line 18 to a on line 18.

21: Not Parallel
         Has IO statement on line 22.
```

Automatic Parallelisation – Example/3

Compiler generated code, part 1:

```
        PROGRAM MAIN
        IMPLICIT NONE
C
C       **** Variables and functions ****
C
        REAL*4 a(1000_8)
        REAL*4 b(1000_8)
        INTEGER*4 i
C
C       **** statements ****
C
C       DOACROSS will be converted to SUBROUTINE __mpdo_MAIN__1
        IF(.NOT. __mp_in_parallel_region$()) THEN
C$DOACROSS shared(a), local(i), shared(a, b)
          DO i = 1, 1000, 1
            a(i) = REAL(i)
            b(i) = a(i)
          END DO
        ELSE
          DO i = 1, 1000, 1
            a(i) = REAL(i)
            b(i) = a(i)
          END DO
        ENDIF
```

Automatic Parallelisation – Example/4

Compiler generated code, part 2:

```
C       DOACROSS will be converted to SUBROUTINE __mpdo_MAIN__2
        IF(.NOT. __mp_in_parallel_region$()) THEN
C$DOACROSS local(i), shared(a, b)
          DO i = 1, 1000, 1
            b(i) = (a(i) + b(i))
          END DO
        ELSE
          DO i = 1, 1000, 1
            b(i) = (a(i) + b(i))
          END DO
        ENDIF
        DO i = 2, 1000, 1
          a(i + -1) = (a(i) + b(i))
        END DO
        DO i = 1, 1000, 1
          PRINT *, 'a(', i, ')=', a(i), ' b(', i, ')=', b(i)
        END DO
        STOP
        END ! MAIN
```

MP Programming on SGI systems

Example Shared Memory Implementation

```
DO i = 1, 1000
   A(i) = 2 * B(i)
END DO
```

```
DO  i = 1, 250
    A(i) = 2 * B(i)
END DO
```
P0

```
DO  i = 251, 500
    A(i) = 2 * B(i)
END DO
```
P1

```
DO  i = 501, 750
    A(i) = 2 * B(i)
END DO
```
P2

```
DO  i = 751, 1000
    A(i) = 2 * B(i)
END DO
```
P3

```
DO i = 2, 999
   B(i)=C(i-1)+C(i+1)
END DO
```

```
DO  i = 2, 251
    B(i)=C(i-1)+C(i+1)
END DO
```
P0

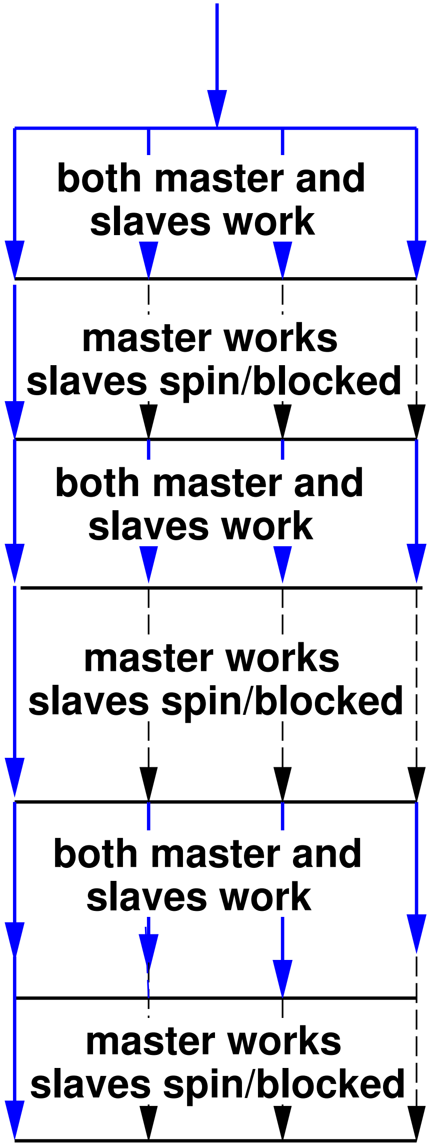
```
DO  i = 252, 501
    B(i)=C(i-1)+C(i+1)
END DO
```
P1

```
DO  i = 502, 750
    B(i)=C(i-1)+C(i+1)
END DO
```
P2

```
DO  i = 751, 999
    B(i)=C(i-1)+C(i+1)
END DO
```
P3

An MP Program on SGI systems

```
program test
<initialize program>

c$doacross
   do i=1,imax
      <do parallel work>
   end do

<do serial work>

c$doacross
   do i=1,imax
      <do parallel work>
   end do

<do serial work>

c$doacross
   do i=1,imax
      <do parallel work>
   end do

<do serial work>
```



master thread
create slaves
enter parallel region

exit parallel region

enter parallel region

exit parallel region

enter parallel region

exit parallel region

Dynamic threads are also supported (more on this to follow)

An MP Program on SGI systems

Slave thread parallel region

```
mp_slave_control(myid) {

while (1) {
      /* wait for next parallel region */
      get_work = __mp_slave_wait_for_work();
      /* either come back with work or after having timed out */
      if (get_work) {

          /* do my portion of the parallel loop:
                • set the uplink frame pointer
                • determine iterations to execute
                • call entry routine routine.pregion0 */
       } else {

       /* go to sleep: will be awakened by master when new work is
          generated */
      }
    }
}
```

Synchronization

Master: __mp_evc_get(), mp_wait_for_loop_completion()
Slaves: __mp_slave_wait_for_work()

An MP Program on SGI systems

Parallel region

Compiler generates micro–tasked routine that contains loop body:

```
routine.pregion0(base, tripcount, lastiter) {
    for (i=base; i<tripcount; i+=uplink–>stride) {
        loop body
    }
}
```

Creating slave threads

```
__mp_do_init()
mp_create()
__mp_check_setup()
mp_crack_env()
```

Master thread parallel region

```
__mp_parallel_do(&pregion0, uplink, schedtype, base, tripcount, stride, chunksize)
```

- Creates slaves if necessary.
- Makes sure all slaves participating in this parallel loop are awake; wakes them up if necessary
- Does master's portion of the parallel loop
- Waits for all slaves to complete their portion of the loop

General Performance Issues

- Parallelisation Overhead
- Load Balancing
- Granularity

Reducing Parallel Overhead

- Parallel overhead is the processing time spent in:
  - Creating slave processes,
  - Starting/ending parallel regions,
  - Synchronization (for example at the end of parallel regions)
  - Extra code added for parallelisation

- Measuring overhead:
  - Time serial run
  - Time the parallelised program on 1 and N processors.

- How to reduce Parallelisation overhead:
  - Parallelise largest possible loops; outermost loops
  - Don't parallelise loops taking a small amount of time
  - Avoid updates of shared scalars (use arrays instead)
  - Use condition modifier in c$doacross:

    c$doacross if (n > *min_n*), ...

Don't use more CPUs than necessary

Reducing Parallel Overhead

Controlling the Number of Threads:

- By default, the number of threads used equals :
  max(1, the number of CPUs on the system–2).

- Environment variables can be used to control the number of threads:

  %setenv MP_SET_NUMTHREADS Nprocs

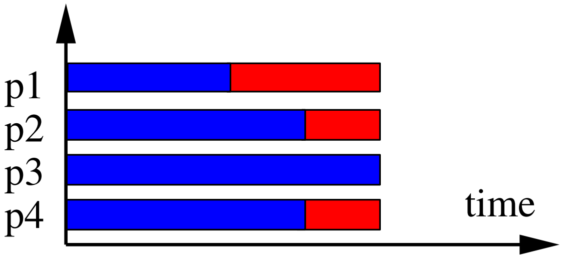
- Always set this number yourself!

- More discussions to follow.

Load Balancing

Load balancing: Evenly dividing the workload among the processors

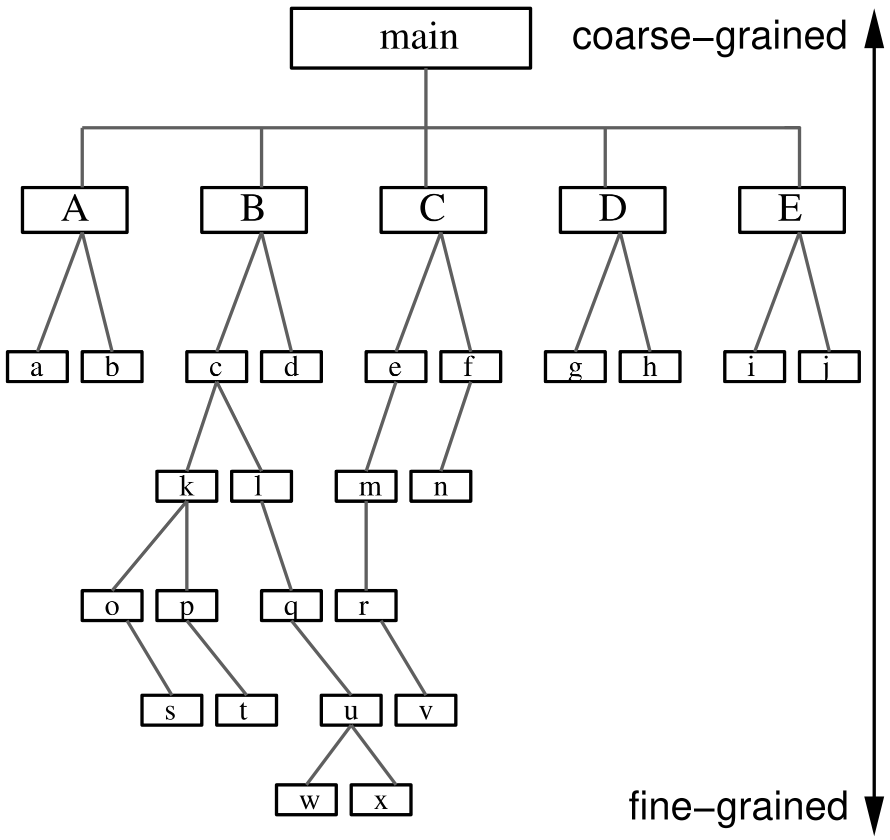
- The time to complete this code segment is limited by the time spent by the longest running thread
- Reducing overhead and balancing the load will improve speedup
- If the following routines
  __mp_evc_get()
  mp_wait_for_loop_completion()
  __mp_slave_wait_for_work()
  end up high in your program's profile, it is likely that you have a load imbalance



- Directive options (e.g., dynamic or gss) provide a convenient way to cure many load imbalance problems

Fine–Grained versus Coarse–Grained

"Granularity" is used to describe the relative size of the parallel region

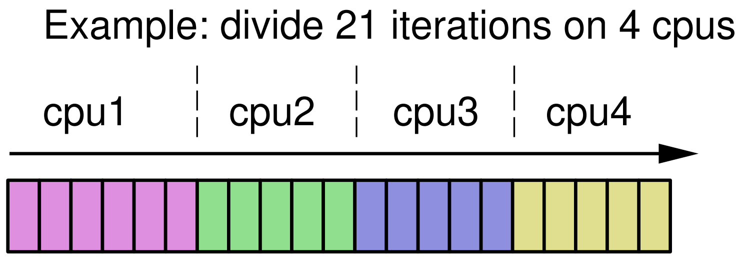


Fine–Grained versus Coarse–Grained

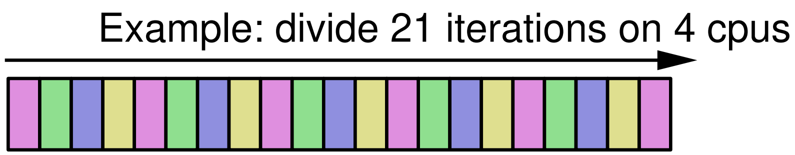
- Fine–grained parallelism (typically loop decomposition)
  - Can be implemented incrementally, one loop at a time
  - Does not require a deep knowledge of the code
  - A lot of loops have to be parallelized to achieve a decent speedup
- Coarse–grained parallelism (typically domain decomposition)
  - Parallelizes larger loops at higher levels, enclosing many smaller loops
  - More code is parallelized
  - Fewer loops are parallelized, reducing overhead
  - Requires deeper knowledge of the code

Iteration Scheduling Options

simple ***(chunk N/A)***



interleave ***(chunk=1 by default)***



dynamic ***(chunk=1 by default)***

Dynamically allocate iterations Analogy: the airline counter queue.

gss ***(guided self–scheduling, chunk N/A)***

Like dynamic scheduling, but start with large chunks, end with small.
Analogy: loading the car trunk, the bin–packing problem..

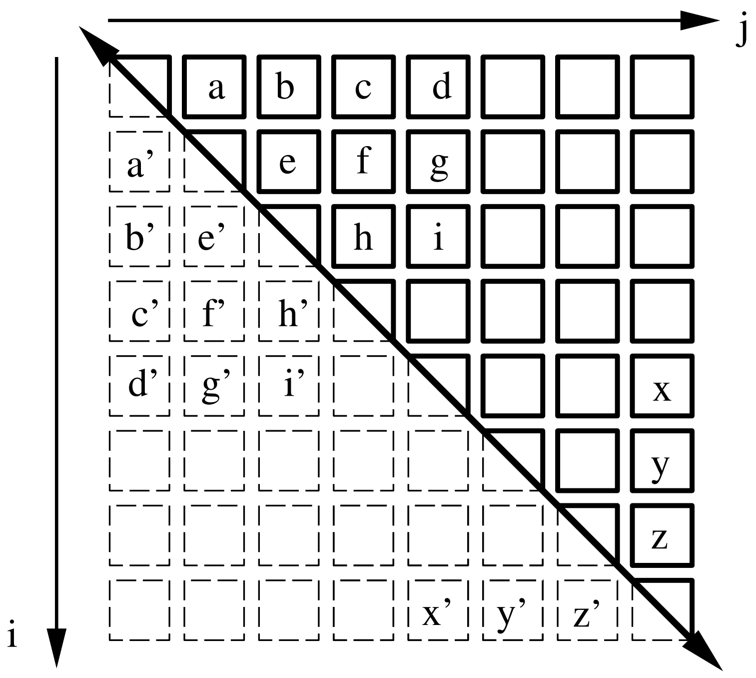
runtime

Environment variables mp_schedtype and chunk are set at runtime

Iteration Scheduling – Example

Example: Matrix Transpose

```
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
```



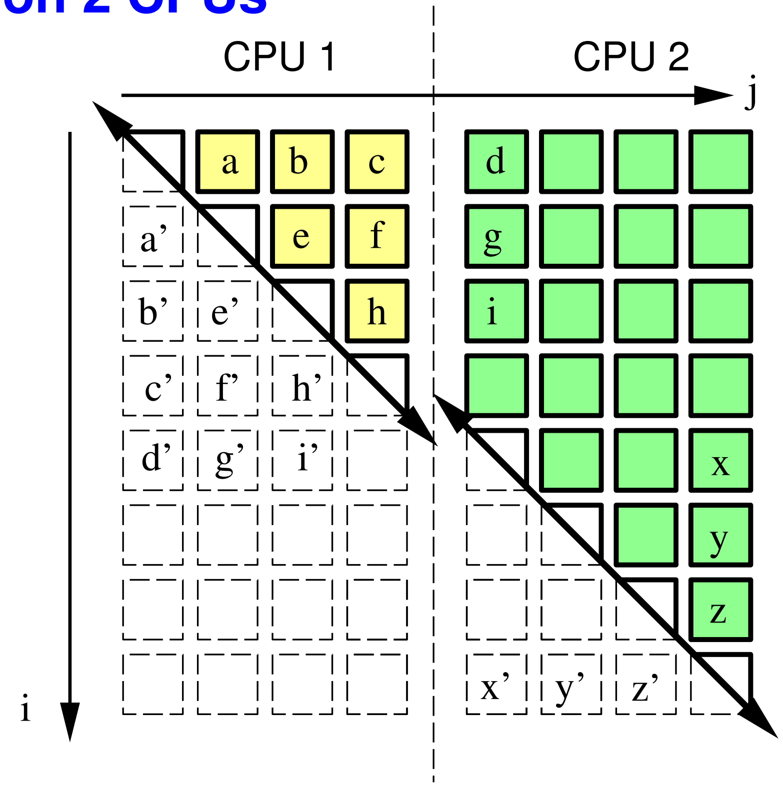
On 2 processors, what is the best mp_schedtype?

(a) simple (b) interleave (c) dynamic (d) gss

Iteration Scheduling – Example

Using mp_schedtype = simple on 2 CPUs

```
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
```



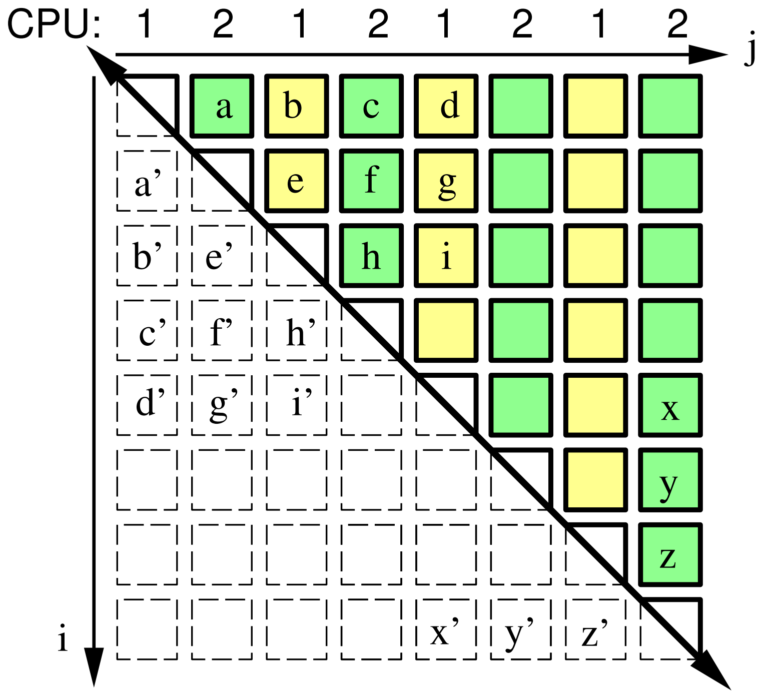
Load balancing is very poor:

cpu 1 transposes 6 pairs of elements
cpu 2 transposes 22 pairs of elements

Iteration Scheduling – Example

Using mp_schedtype = interleave, chunk=1 on 2 CPUs

```
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
```



Load balancing is improved, still not perfect:

CPU 1 transposes 12 pairs of elements
CPU 2 transposes 16 pairs of elements

Dynamic or gss may give better balancing

(but I haven’t tried this yet)

Load Balancing

- **Why so many mp_schedtypes?**
  For optimal load balance.

- **Why balance the load?**
  Performance is limited by the slowest thread.

- **Finding load imbalance:**
  Profile cputime for threads.

- **Choose best scheduling for the most time–consuming loops**
  Experiment using environment variables; can hard–code the best choice.

Iteration Scheduling Controls

How to specify a non–default scheduling type

- **Through a compiler option**

  –mp_schedtype = simple | interleave | dynamic | gss | runtime

  be aware that this only takes effect in combination with the (auto–) parallelization options –pfa/–pca or –mp

- **Specify the desired scheduling type with a directive**
  - **As part of the *C$DOACROSS* directive**
  - **As part of the *C$PAR PARALLEL DO* PCF directive**
  - **Use a *C$MP_SCHEDTYPE=...* and *C$CHUNK=...* directive**
  - **As part of the *#pragma pfor* pragma**

More details to follow later

Environment Variables – 1

The following act like implicit calls to the corresponding MP library routines:

% setenv MP_SET_NUMTHREADS N

% setenv MP_BLOCKTIME N

Set N = 0 to spin forever!

% setenv MP_BLOCKTYPE *SLEEP* | YIELD

To specify scheduling type and chunk size at runtime which have been specified with c$doacross mp_schedtype=runtime:

% setenv MP_SCHEDTYPE type

% setenv CHUNK N

To control gang scheduling:

% setenv MP_GANG *ON* | OFF

Environment Variables – 2

The size of the stack (which is used for local variables) has a default of 16 MByte for up to 64 processors (4MB for more than 64 processors)

You can change this as follows:

% setenv MP_SLAVE_STACKSIZE *"stacksize_in_bytes"*

Dynamic Threads

Using Dynamic Threads:

MP_SUGNUMTHD (suggested number of threads)
MP_SUGNUMTHD_MIN (minimum)
MP_SUGNUMTHD_MAX (maximum)
MP_SUGNUMTHD_VERBOSE (verbosity)

Please note that:

- ***Thread adjustment is done BEFORE a parallel region is entered***
- ***Usage of mp_set_numthreads() and mp_numthreads() will disable usage of dynamic threads (safety precaution)***

If you want to look at how many threads you get, use mp_suggested_numthreads(0)

Using "0" as the argument which returns the current value without changing it. Using any other positive number as the argument will request (but not require) that number of threads.

The MP Utility Library

Thread information

mp_set_numthreads(num)
imp_numthreads()
mp_my_threadnum()
mp_in_doacross_loop()
mp_suggested_numthreads()

Synchronization

mp_setlock()
mp_unsetlock()
mp_barrier()

Slave threads sleep control

mp_block()
mp_unblock()
mp_blocktime(iters)

Thread management

mp_setup()
mp_create(num)
mp_destroy()
mp_set_slave_stacksize(stacksize_in_bytes)

Also "atomic" fetch&op operations are available through a low-level library. One can use these as well. For more info: "man fetchop"

Documentation: % man 3 mp

The MP Library and Portability

- Straightforward usage of the MP library violates portability

- Therefore one can use:

  1. *#ifdef preprocessor constructs*

  2. *the fact that with the –mp option, the compiler will recognize comments that start with C$*

Timing and profiling MP programs

- **To obtain timing information for an MP program:**

```
% setenv MP_SET_NUMTHREADS ....
% time ./a.out
```

- **Use SpeedShop to obtain profiling information for an MP program:**

```
% setenv MP_SET_NUMTHREADS ....
% ssrun –exp pcsamp ./a.out

% foreach file ( a.out.pcsamp.* )
      prof [–lines] $file
    end
```

This gives one profile for each thread

```
% prof [–lines] a.out.pcsamp.*
```

This gives one profile for all threads

Example – Time command

Using the time command to measure performance:

```
ruud@eo2k1> setenv MP_SET_NUMTHREADS 1
ruud@eo2k1> time ./CWIROS  > cwiros.res
9.678u 0.091s 0:09.86 98.9% 0+0k 0+1io 0pf+0w

ruud@eo2k1> setenv MP_SET_NUMTHREADS 2
ruud@eo2k1> time ./CWIROS  > cwiros.res
10.363u 0.121s 0:05.72 183.2% 0+0k 3+8io 3pf+0w
```

From this we derive the following performance data

Processors	User+System	Elapsed
1	9.769	9.86
2	10.484	5.72

Conclusions: Parallel overhead is 7.3%

Parallel speed–up is 1.72

Example – SpeedShop

Using SpeedShop to analyze performance:

```
ruud@eo2k1> setenv MP_SET_NUMTHREADS 2
ruud@eo2k1> ssrun -exp pcsamp ./CWIROS > cwiros.res

ruud@eo2k1> prof CWIROS.pcsamp.12668 >& cwiros_t1.pro
ruud@eo2k1> prof CWIROS.pcsamp.12674 >& cwiros_t2.pro

ruud@eo2k1> prof CWIROS.pcsamp.* >& cwiros_2procs.pro
```

Example – MP SpeedShop Output

```
samples    time(%)       cum time(%)      procedure (dso:file)

    103      1s( 20.6)      1s( 20.6)         divfree (./CWIROS:DIVFREE.f)
     96  0.96s( 19.2)      2s( 39.7)          ROWKAP (./CWIROS:KAPNEW.f)
     67  0.67s( 13.4)   2.7s( 53.1)            ITER (./CWIROS:CHEMIE.f)
     40   0.4s(  8.0)   3.1s( 61.1)           FLUXY (./CWIROS:KAPNEW.f)
     35  0.35s(  7.0)   3.4s( 68.1)          __expf ( .. fexp.c)
     30   0.3s(  6.0)   3.7s( 74.1) __mp_barrier_nthreads (..libmp.so)
     26  0.26s(  5.2)     4s( 79.2)           FLUXX (./CWIROS:KAPNEW.f)
     24  0.24s(  4.8)   4.2s( 84.0)           KAPPA (./CWIROS:KAPNEW.f)
     23  0.23s(  4.6)   4.4s( 88.6) __mp_slave_wait_for_work (.. libmp.s)
     11  0.11s(  2.2)   4.5s( 90.8)         TWOSTEP (./CWIROS:CHEMIE.f)
.......................................................................
    124   1.2s( 24.0)   1.2s( 24.0)         divfree (./CWIROS:DIVFREE.f)
     93  0.93s( 18.0)   2.2s( 42.1)          ROWKAP (./CWIROS:KAPNEW.f)
     76  0.76s( 14.7)   2.9s( 56.8)            ITER (./CWIROS:CHEMIE.f)
     36  0.36s(  7.0)   3.3s( 63.8)           FLUXY (./CWIROS:KAPNEW.f)
     27  0.27s(  5.2)   3.6s( 69.0)          __expf (.. libm/fexp.c)
     23  0.23s(  4.5)   3.8s( 73.4)           FLUXX (./CWIROS:KAPNEW.f)
     22  0.22s(  4.3)     4s( 77.7)           KAPPA (./CWIROS:KAPNEW.f)
     15  0.15s(  2.9)   4.2s( 80.6) __mp_barrier_master (.. libmp.so)
     15  0.15s(  2.9)   4.3s( 83.5)            diff (./CWIROS:DIFF.f)
      9  0.09s(  1.7)   4.4s( 85.3)         TWOSTEP (./CWIROS:CHEMIE.f)
      9  0.09s(  1.7)   4.5s( 87.0)          INJECT (./CWIROS:GRIDFV.f)
      8  0.08s(  1.6)   4.6s( 88.6)           depos (./CWIROS:DEPOS.f)
      8  0.08s(  1.6)   4.6s( 90.1)            NORM (./CWIROS:CWIROS.f)
      5  0.05s(  1.0)   4.7s( 91.1)         LOOP991 (./CWIROS:CHEMIE.f)
```

Debugging Parallel Code

Don't write bugs (if you must, then know where you put them)

Seriously, finding bugs in parallel code is hard, no matter how good the debugger. Implement parallelism very carefully, with frequent testing, so when you break the code, you know where to look already. Tip: use RCS.

Testing steps:

1. **Test any changes with serial compilation to confirm general correctness of code without –mp**
2. **Compile with –mp and test on 1 processor**
   Problem that surfaces here is declaring a variable *local* when it is not, and really has scope outside of the loop.
3. **Test on N (=2,3,4,....) processors**
   Problems that surface here are data dependencies that were not detected and fixed, and variables declared *shared* that are really *local* so the threads are using the same location for their scratch calculations. Race condition bugs may not show up on every run.

Shared Memory Parallelisation

About shared memory parallelization

- Shared Memory does not mean that all data is actually shared
- There is often a need for "local" data as well
- Take the following example:

```
for (i=0; i<10; i++)
      a[i] = b[i] + c[i];
```

- Let's assume we run this on 2 processors:
  processor 1 for i=0,2,4,6,8
  processor 2 for i=1,3,5,7,9

Shared Memory Parallelisation

Processor 1

```
for i1 = 0, 2, 4, 6, 8 do:
    a[i1] = b[i1] + c[i1];

Read:  b,c     Write:   a
```

Processor 2

```
for i2 = 1, 3, 5, 7, 9 do:
    a[i2] = b[i2] + c[i2];

Read:  b,c     Write:   a
```

Shared Memory Parallelisation

Processor 1

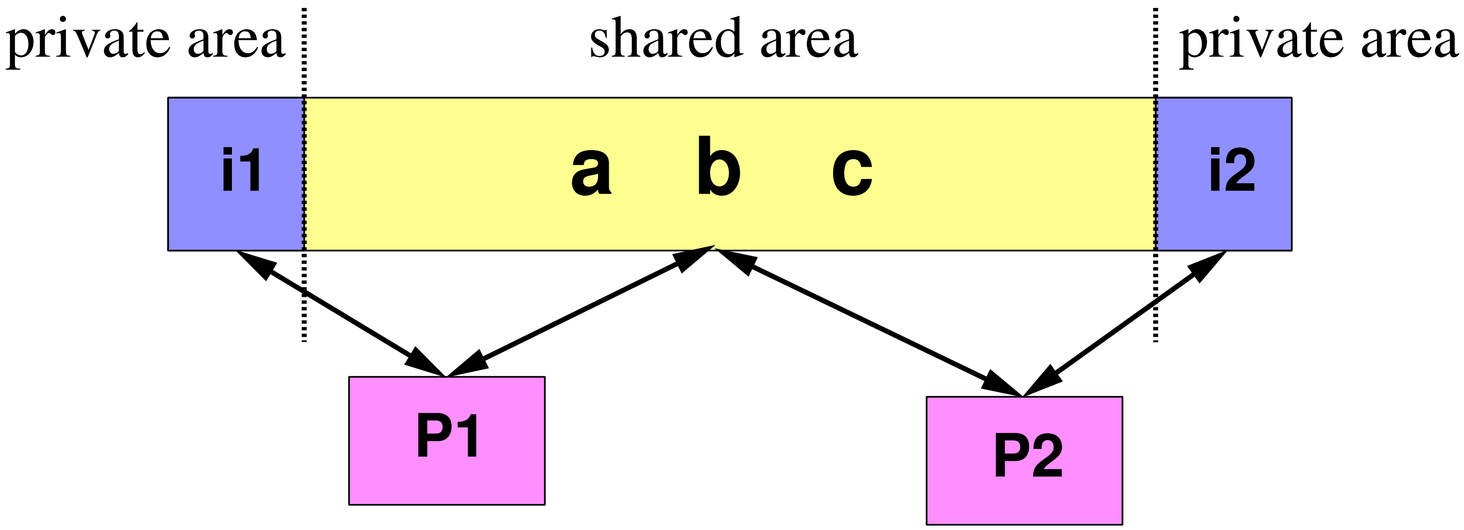
```
for i1 = 0, 2, 4, 6, 8 do:
    a[i1] = b[i1] + c[i1];
```

Read: b,c   Write: a

Processor 2

```
for i2 = 1, 3, 5, 7, 9 do:
    a[i2] = b[i2] + c[i2];
```

Read: b,c   Write: a



Example: How to use local data

```
sum    = 0.0
 do j=1,jmax
      do i = 1,imax
           sum    = sum + a(i,j)
      end do
end do
```

```
           sum = 0.0
c$doacross local(i,j,sumj)
c$&              shared(jmax, imax, sum, a)
           do j=1,jmax
                sumj = 0.0
                do i=1,imax
                     sumj = sumj + a(i,j)
                end do
c$              call mp_setlock()
                sum = sum + sumj
c$              call mp_unsetlock()
           end do
```

Example: Reduction hand coded

```
sum = 0
```

Processor 1

```
c$doacross (....)
   for j = 1, jmax/2
      sumj = 0

      for i = 1, imax
           sumj = sumj+a(i,j)
      end for
   end for
c$ call mp_setlock()
      sum = sum + sumj
c$ call mp_unsetlock()
```

idle

Processor 2

```
c$doacross (....)
   for j = jmax/2+1, jmax
      sumj = 0

      for i = 1, imax
           sumj = sumj+a(i,j)
      end for
   end for
```

idle

```
c$ call mp_setlock()
      sum = sum + sumj
c$ call mp_unsetlock()
```

synchronize processors

Shared Memory Parallelisation

A Question

```
for (i=0; i<10; i++)
    a[i] = b[i] + c[i];
```

Let's assume we run this on 2 processors:

processor 1	for	i=0,2,4,6,8
processor 2	for	i=1,3,5,7,9

This is <u>not</u> an efficient way to do this!

WHY?

Shared Memory Parallelisation

The Answer

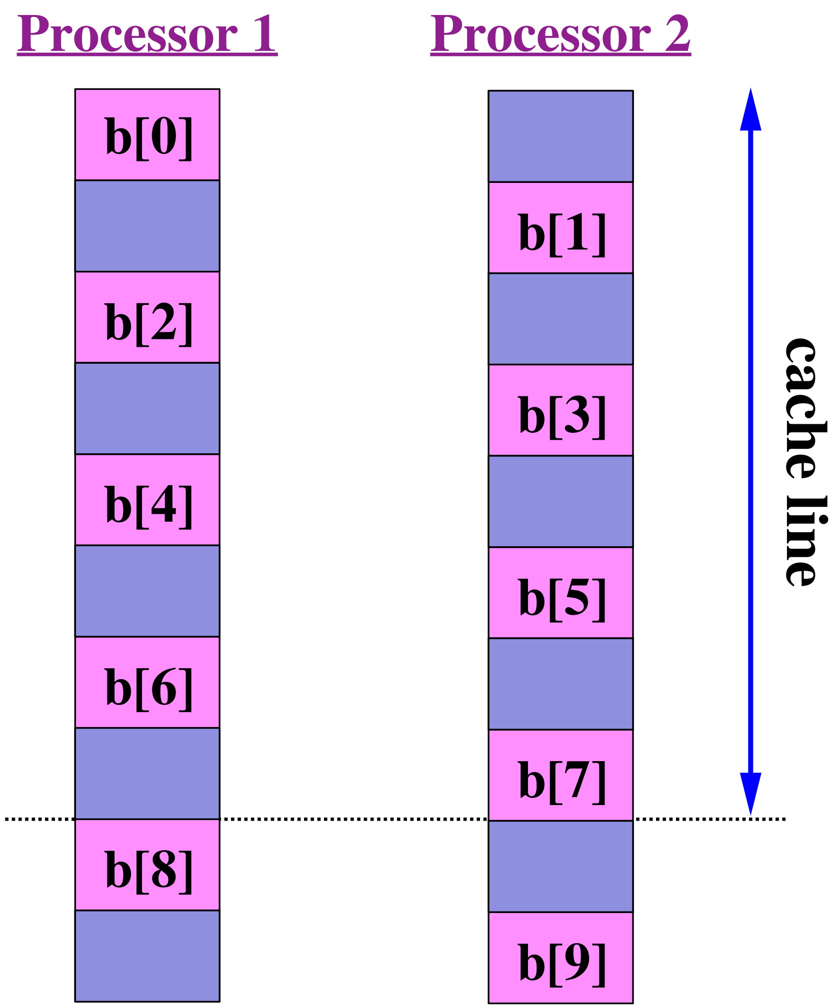
```
for (i=0; i<10; i++)
    a[i] = b[i] + c[i];
```

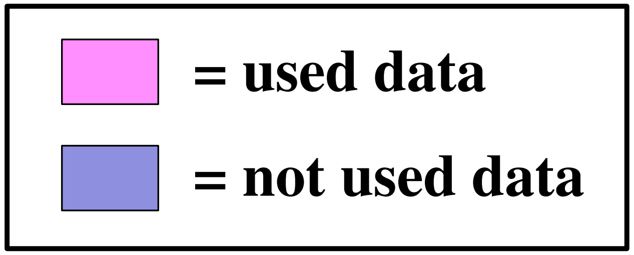
Because of cache line usage

b[] and c[]: we use half of the data

a[]: false sharing

Poor cache line utilisation





Both processors read the same cache lines

Note: the same argument holds for array c[]

False Sharing

Processor 1

a[0] = b[0] + c[0]

Write into the line containing a[0]

This marks the cache line containing a[0] as "dirty"

Processor 2

time

False Sharing

Processor 1

a[0] = b[0] + c[0]

Write into the line containing a[0]

This marks the cache line containing a[0] as "dirty"

Processor 2

a[1] = b[1] + c[1]

Detects the line with a[1] is dirty

Get a fresh copy (from processor 1)

Write into the line containing a[1]

This marks the cache line containing a[1] as "dirty"

time

False Sharing

Processor 1

a[0] = b[0] + c[0]

Write into the line containing a[0]

This marks the cache line containing a[0] as "dirty"

Processor 2

a[1] = b[1] + c[1]

Detects the line with a[1] is dirty

Get a fresh copy (from processor 1)

Write into the line containing a[1]

This marks the cache line containing a[1] as "dirty"

Processor 1

a[2] = b[2] + c[2]

Detects the line with a[2] is dirty

Get a fresh copy (from processor 2)

Write into the line containing a[2]

This marks the cache line containing a[2] as "dirty"

time

False Sharing

Processor 1

a[0] = b[0] + c[0]

Write into the line containing a[0]

This marks the cache line containing a[0] as "dirty"

a[2] = b[2] + c[2]

Detects the line with a[2] is dirty

Get a fresh copy (from processor 2)

Write into the line containing a[2]

This marks the cache line containing a[2] as "dirty"

Processor 2

a[1] = b[1] + c[1]

Detects the line with a[1] is dirty

Get a fresh copy (from processor 1)

Write into the line containing a[1]

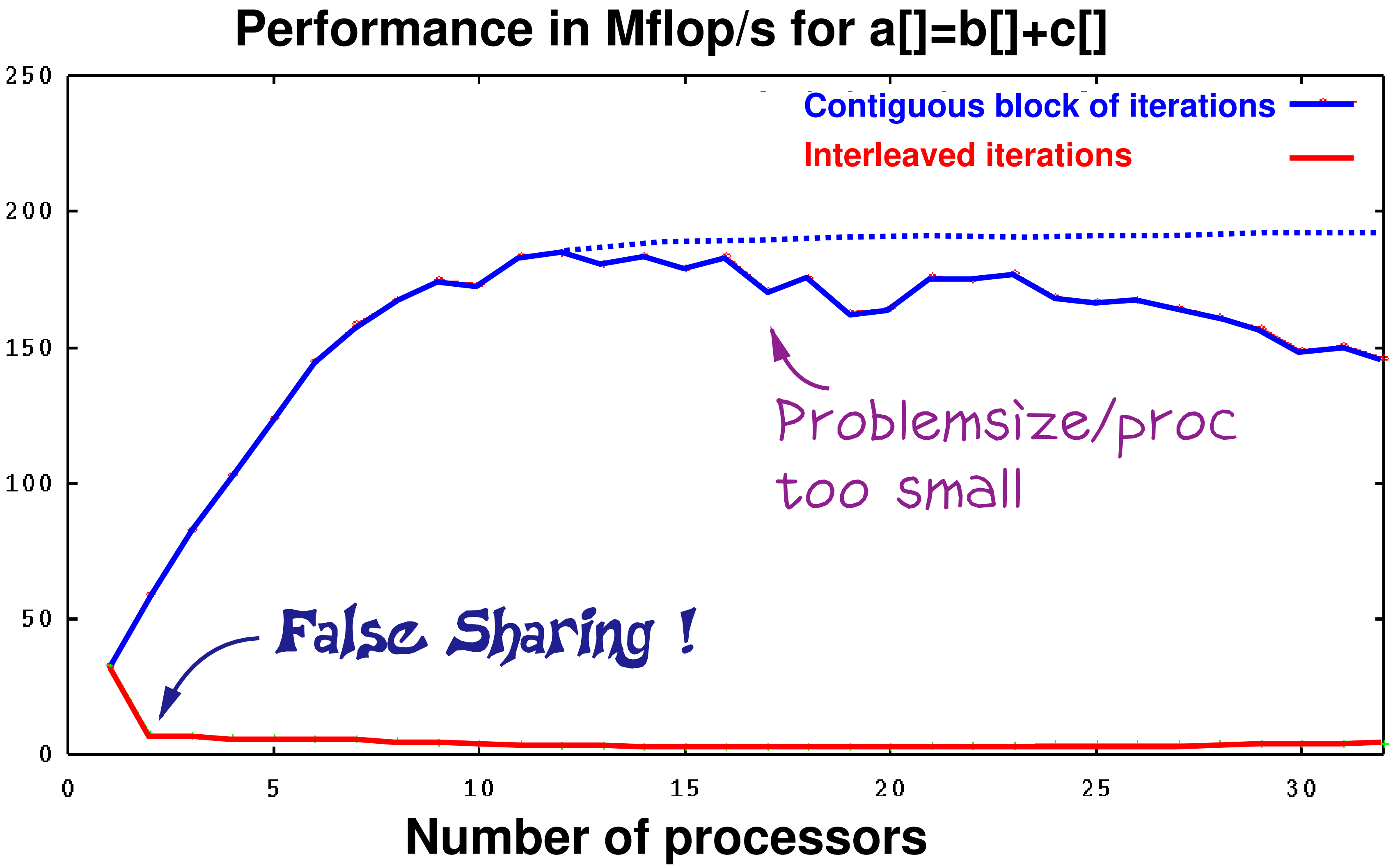
This marks the cache line containing a[1] as "dirty"

a[3] = b[3] + c[3]

Detects the line with a[3] is dirty

time

False Sharing – O2000 Performance



Cache lines in Parallel Codes

Lessons Learned:

- Use full cache lines where possible

  Be aware of non–unit memory access

  The compilers are aware of this

- Avoid false sharing

  Potentially can occur with shared data read&write data

  The compiler will try to avoid it
  and not distribute the loop in the way just shown!

Parallelisation Strategy

0) port code if necessary. Make sure it gives correct answer.
1) (f)split code files for easy access to individual modules
2) profile the code, work only on loops and routines that use a large percentage of cpu time (think about coarse–grained parallelism)
3) use PFA and be done (!?), or
4) hand tune for loop level parallelization
    - tune the code to improve serial performance
    - if the loop to be parallelized contains procedure calls, check shared data, remove auto–save constructs and avoid aliasing statements. Fortran: don't use compiler option –static
    - modify the code to remove data dependencies
    - insert c$doacross/#pragma parallel directives, and compile the code with –mp
    - run the code, measure speedup
    - compare output to ensure the correctness
    - repeat the above 5 bullets until done. Use PFA and DeveloperMagic to help.
5) To further improve performance, look for coarse–grained parallelism opportunities.