Practical Issues in OpenMP

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

High Performance Computing I, 2009

Loop Scheduling

- The way in which iterations of a parallel loop get assigned to threads is determined by the loop's **schedule**
- Default scheduling typically assumes an equal load balance, frequently the case that different iterations can have entirely different computational loads
- Load imbalance can cause significant synchronization delays

Static vs. Dynamic Scheduling

Basic distinction of loop scheduling:

Static:	iteration assignment to threads determined as function of iteration/thread number
Dynamic:	assignment can vary at run-time, and iterations are handed out to threads as they complete previously assigned iterations

- Iterations in both schemes can be assigned in **chunks**

SCHEDULE Clause

The general form of the SCHEDULE clause:

SCHEDULE clause

schedule(*type*[,chunk])

where *type* can be one of:

static	without `chunk`, threads given equally sized subdivision of iterations (exact placement implementation-dependent). With `chunk`, iterations divided into `chunk`-sized pieces, remainder allocation is implementation dependent
dynamic	iterations divided into chunks (default is one if `chunk` not present), assigned dynamically at run-time

guided	first chunk size determined by implementation, then subsequently decreased exponentially (value is implementation-dependent) to minimum size specified by `chunk` (default 1)
runtime	`chunk` must not appear, schedule determined by value of environmental variable `OMP_SCHEDULE`
auto	(OpenMP 3.0) gives implementation freedom to choose best mapping of iterations to threads

Scheduling Considerations

Things to consider when choosing between scheduling options

- Dynamic schedules can better balance the load between threads, but typically have higher overhead costs (synchronization costs per chunk)
- Guided schedules have the advantage of typically requiring fewer chunks (translates to fewer synchronizations) - typically the initial chunk size is roughly the number of iterations divided by the number of threads
- Simple static has the lowest overhead, but is most susceptible to load imbalances

Easy to Use?

- OpenMP does not force the programmer to explicitly manage communication or how the program data is mapped onto individual processors - sounds great ...
- OpenMP program can easily run into common SMP programming errors, usually from resource contention issues.

Directive Nesting

- DO/for, SECTIONS, SINGLE, and WORKSHARE directives that bind to the same parallel region are not allowed to be nested.
- DO/for, SECTIONS, SINGLE, and WORKSHARE directives are not allowed in the dynamical extent of CRITICAL, ORDERED, and MASTER directives.
- BARRIER and MASTER are not permitted in the dynamic extent of DO/for, SECTIONS, SINGLE, WORKSHARE, MASTER, CRITICAL, and ORDERED directives.
- ORDERED must appear in the dynamical extent of a DO or PARALLEL DO with an ORDERED clause. ORDERED is not allowed in the dynamical extent of SECTIONS, SINGLE, WORKSHARE, CRITICAL, and MASTER.

Data Storage Defaults

- Most variables are SHARED by default

  Fortran: `common` blocks, `save` variables , `MODULE` variables.

  C: file scope variables, static variables.

- with some exceptions ...
  - stack variables in sub-programs called from a PARALLEL region.
  - automatic variables within a statement block
  - loop indices (in C just on “work-shared” loops)

Data Storage Gotchas

- Assumed size and assumed shape arrays can not be privatized.
- Fortran allocatable arrays (and pointers) can be PRIVATE or SHARED, but not FIRSTPRIVATE or LASTPRIVATE.
- Constituent elements of a PRIVATE (FIRSTPRIVATE/LASTPRIVATE) name common block can not be declared in another data scope clause.
- Privatized elements of shared common blocks are no longer storage equivalent with the common block.

Synchronization Awareness

Implied Barriers :

1. `END PARALLEL`
2. `END DO` (unless NOWAIT)
3. `END SECTIONS` (unless NOWAIT)
4. `END CRITICAL`
5. `END SINGLE` (unless NOWAIT)

Implied Flushes :

1. `BARRIER`
2. `CRITICAL/END CRITICAL`
3. `END DO`
4. `END PARALLEL`
5. `END SECTIONS`
6. `END SINGLE`
7. `ORDERED/END ORDERED`

Synchronization Costs

- Overhead for synchronization on an SGI Origin 2000 (MIPS 250MHz R10000 processors)

Nthreads	PARALLEL[μs]	DO[μs]	ATOMIC[μs]	REDUCTION[μs]
1	2.0	2.3	0.1	2.1
2	8.4	7.8	0.4	11.0
4	11.6	6.8	1.5	20.7
8	28.0	14.1	3.1	31.0

- 10μs? Isn't that pretty small?
- 10μs$\times$250MHz =**2500 clock cycles** - lost computation.

Synchronization Costs (cont'd)

- Overhead for synchronization on an SGI Altix 3700 (Intel 1300MHz Itanium2 processors)

Nthreads	PARALLEL[μs]	DO[μs]	ATOMIC[μs]	REDUCTION[μs]
1	0.3	0.3	0.1	0.5
2	2.3	2.1	0.4	2.6
4	5.9	4.7	0.4	9.6
8	6.6	6.8	0.5	24.1
16	10.3	10.7	0.6	60.7
32	19.2	19.3	0.7	132
64	41.8	40.9	0.7	316

- 10μs? Isn't that pretty small?
- 10μs$\times$1300MHz =**13000 clock cycles** - lost computation.
- Not exactly great progress ...

Synchronization Costs (cont'd)

- Overhead for synchronization on an Intel "Clovertown" (dual quad-core 1.866GHz Xeon processors)

Nthreads	PARALLEL[μs]	DO[μs]	ATOMIC[μs]	REDUCTION[μs]
1	0.2	0.2	0.02	0.2
2	1.6	1.7	0.08	2.0
4	2.3	2.4	0.14	3.1
8	3.8	3.9	0.52	5.8

- 5.8μs$\times$1866MHz =**10823 clock cycles** - lost computation.
- Overhead for synchronization on an Intel "Nehalem" (dual quad-core 2.8GHz Xeon processors)

Nthreads	PARALLEL[μs]	DO[μs]	ATOMIC[μs]	REDUCTION[μs]
1	0.1	0.1	0.01	0.1
2	1.1	1.1	0.04	1.2
4	1.2	1.2	0.05	1.5
8	1.7	1.8	0.05	2.5

- 2.5μs$\times$2800MHz =**7000 clock cycles** - lost computation.

Common Errors

Race conditions	: outcome of the program depends on detailed scheduling of thread team (the answer is different every time I run the code!).
Deadlock	: threads wait forever for a locked resource to become free.

Race Conditions

- What is wrong with this code fragment?

```
      real tmp,x
!$OMP PARALLEL DO REDUCTION(+:x)
      do i=1,10000
         tmp=dosomework(i)
         x=x+tmp
      end do
!$OMP END DO
      y(iam) = work(x,iam)
!$OMP END PARALLEL
```

Race Conditions

- What is wrong with this code fragment?

```
      real tmp,x
!$OMP PARALLEL DO REDUCTION(+:x)
      do i=1,10000
         tmp=dosomework(i)
         x=x+tmp
      end do
!$OMP END DO
      y(iam) = work(x,iam)
!$OMP END PARALLEL
```

- The programmer did not make tmp PRIVATE, hence the results are unpredictable.

Race Conditions

- What about now?

```
      real tmp,x
!$OMP PARALLEL DO REDUCTION(+:x),PRIVATE(tmp)
      do i=1,10000
         tmp=dosomework(i)
         x=x+tmp
      end do
!$OMP END DO NOWAIT
      y(iam) = work(x,iam)
!$OMP END PARALLEL
```

Race Conditions

- What about now?

```
      real tmp,x
!$OMP PARALLEL DO REDUCTION(+:x),PRIVATE(tmp)
      do i=1,10000
         tmp=dosomework(i)
         x=x+tmp
      end do
!$OMP END DO NOWAIT
      y(iam) = work(x,iam)
!$OMP END PARALLEL
```

- The value of x is not dependable without the barrier at the end of the DO construct - be careful with NOWAIT!

Deadlock

- A somewhat artificial example of deadlock - watch that resources are freed if you are using locks!

```
      call OMP_INIT_LOCK(lock0)
!$OMP PARALLEL SECTIONS
!$OMP SECTION
      call OMP_SET_LOCK(lock0)
      iret = dolotsofwork()
      if (iret.le.tol) then
         call OMP_UNSET_LOCK(lock0)
      else
         call error(iret)
      endif
!$OMP SECTION
      call OMP_SET_LOCK(lock0)
      call compute(A,B,iret)
      call OMP_UNSET_LOCK(lock0)
$!OMP END SECTIONS
```

Load Balancing

- Consider the following code fragment - can you see why it not efficient to parallelize on the outer loop?

```
do i=1,N
   do j=1,i
      a(j,i)=a(j,i)+b(j)*c(i)
   end do
end do
```

Load Balancing

- One strategy - break up the loop into interleaved chunks,

```
!$OMP PARALLEL SHARED (num_threads)
!$OMP SINGLE
      num_threads = OMP_GET_NUM_THREADS()
!$OMP END SINGLE NOWAIT
!$OMP END PARALLEL
!$OMP PARALLEL DO PRIVATE(i,j,k)
      do k = 1, num_threads
         do i = k, n, num_threads
            do j = 1,i
               a(j,i) = a(j,i) + b(j)*c(j)
            end do
         end do
      end do
```

Load Balancing

- Another equivalent (and somewhat cleaner!) way,

```
!$OMP PARALLEL DO PRIVATE(i,j) SCHEDULE(static,4)
      do i=1,n
         do j=1,i
            a(j,i)=a(j,i)+b(j)*c(j)
         end do
      end do
```

Toward Coarser Grains

What is wrong with fine grain (loop) parallelism?

- Overhead kills performance
- Not scalable to large number of threads

$$S(N_p) = \frac{\tau_s + \tau_p}{\tau_s + \tau_p/P} = \frac{1}{S + (1 - S)/P}$$

Remember Amdahl's law!

Coarsening

Strategies for increasing OpenMP performance,

- do more work per parallel region, and decrease fraction of time spent in sequential code.
- reduce synchronization across threads
- combine multiple parallel do directives into larger parallel region (with work-sharing constructs therein)

Coarsening (cont'd)

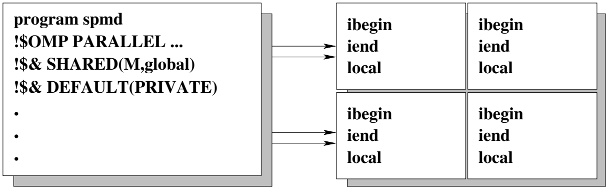
Domain Decomposition

- Break Data domain into sub-domains,
- Compute loop bounds once depending on number of threads (a priori loop decomposition),
- Reduces loop overhead, but shifts burden from compiler back to the programmer,
- Implements the **S**ingle **P**rogram **M**ultiple **D**ata model (**SPMD**).

Coarse Grain SPMD Example

```
      program spmd
$!OMP PARALLEL DEFAULT(PRIVATE) SHARED(N,global)
      num_threads = OMP_GET_NUM_THREADS()
      iam = OMP_GET_THREAD_NUM()
      ichunk = N/num_threads
      ibegin = iam*ichunk
      iend = ibegin + ichunk - 1
      call lotsofwork(ibegin,iend,local)
$!OMP ATOMIC
      global = global + local
$!OMP END PARALLEL
      print*, global
      end program spmd
```

Coarse Grain SPMD Example



SPMD Implementation

- Manual decomposition - valid for any number of threads (make sure that cost/benefit ratio is high enough!)
- Same program on each thread, but a different (PRIVATE) sub-domain of the program data.
- Synchronization necessary to handle global variable updates (ATOMIC usually more efficient than CRITICAL).

Advantages over Message Passing

- Domain decomposition methodology is the same, but implementing it in OpenMP can be easier, as global data can be read without any need for synchronization or message passing.
- Parallelize only parts of the code that require it (profiling is key!). Pre and Post Processing can be left sequential.

Best of Both Worlds?

How about combining OpenMP with Message Passing?

- Message Passing between machines, OpenMP within.
- Allow application dependent mixing within an SMP.
- Coarse grain with Message Passing, fine grain with OpenMP.

Platforms & Compilers

This table lists the various compiler suites available on the production computing platforms along with their OpenMP compiliance:

Platform	Compiler	OMP	Invocation
Linux IA64	Gnu (g77/gcc/g++)	No	–
	Intel (ifort/icc/icpc)	2.5	-openmp -openmp_report2
Linux x86_64	Gnu[a] (g77/gcc/g++)	2.5(>4.1)	–
	PGI (pgf90/pgcc/pgCC)	2.5	-mp
	Intel (ifort/icc/icpc)	2.5,3.0($\geq$ 11.0)	-openmp -openmp_report2

[a]The Gnu compiler suite supports OpenMP for versions >4.2, although some Linux distributions (e.g. RedHat) have backported support to 4.1

Simple OpenMP example

```fortran
program simple
  USE omp_lib ! comment out for pgf90 — if not openmp 2.0 compliant
  implicit none

  integer :: myid, nthreads, nprocs
  !include this declaration for pgf90
  !integer :: OMP_GET_NUM_THREADS,OMP_GET_THREAD_NUM,OMP_GET_NUM_PROCS

!$OMP PARALLEL default(none) private(myid) &
!$OMP shared(nthreads,nprocs)
!
! Determine the number of threads and their id
!
myid = OMP_GET_THREAD_NUM()
nthreads = OMP_GET_NUM_THREADS();
nprocs = OMP_GET_NUM_PROCS();
!$OMP BARRIER
if (myid==0) print*, 'Number of available processors: ',nprocs
print*,'myid = ', myid, ' nthreads ', nthreads
!$OMP END PARALLEL
end program simple
```

Altix - simple example

```
[jonesm@lennon ~/d_omp]$ module load intel
[jonesm@lennon ~/d_omp]$ ifort -O3 -o simple_ifort -openmp -openmp_report2
simple.f90
simple.f90(19) : (col. 6) remark: OpenMP multithreaded code generation BARRIER
was successful.
simple.f90(9) : (col. 6) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
[jonesm@lennon ~/d_omp]$ setenv OMP_NUM_THREADS 4
[jonesm@lennon ~/d_omp]$ ./simple_ifort
 myid =            1  nthreads           4
 myid =            3  nthreads           4
 myid =            2  nthreads           4
 Number of available processors:           4
 myid =            0  nthreads           4
```

U2 - simple example

```
[jonesm@bono ~/d_omp]$ module load intel
[jonesm@bono ~/d_omp]$ ifort —O3 —o simple_ifort —openmp simple.f90
[jonesm@bono ~/d_omp]$ setenv OMP_NUM_THREADS 4
[jonesm@bono ~/d_omp]$ ./simple_ifort
 Number of available processors:           4
 myid =             1   nthreads           4
 myid =             0   nthreads           4
 myid =             2   nthreads           4
 myid =             3   nthreads           4
```

```
[jonesm@bono ~/d_omp]$ module load pgi
[jonesm@bono ~/d_omp]$ pgf90 —O3 —mp —o simple_pgi simple.f90
[jonesm@bono ~/d_omp]$ ./simple_pgi
 Number of available processors:            4
 myid =              0   nthreads            4
 myid =              3   nthreads            4
 myid =              1   nthreads            4
 myid =              2   nthreads            4
```

MD Sample Code

Let’s take this as a trial of parallelizing a real code:

- Take the sample MD code from `www.openmp.org`
- Modify it slightly for our environment (uncomment the line for `use omp_lib`, add conditional compilation for the API function calls ...
- Then do a quick profile to see where the code spends is spending time ...

```
[jonesm@lennon ~/d_omp]$ ifort -O3 -o md.pg -g -p md.f90
[jonesm@lennon ~/d_omp]$ /usr/bin/time ./md.pg
November  5 2005   3:39:39.245 PM

MD
  A molecular dynamics program.



   100    124395.       0.226163       0.162282E-05
   200    124395.       0.918574       0.659101E-05
   300    124395.        2.07756       0.149066E-04
   400    124395.        3.70360       0.265724E-04
   500    124395.        5.79733       0.415922E-04
   600    124395.        8.35961       0.599709E-04
   700    124394.        11.3914       0.817147E-04
   800    124394.        14.8940       0.106831E-03
   900    124393.        18.8688       0.135327E-03
  1000    124393.        23.3172       0.167213E-03

MD
  Normal end of execution.

November  5 2005   3:40:50.247 PM
70.94user 0.00system 1:11.23elapsed 99%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (116major+120minor)pagefaults 0swaps
```

```
[jonesm@lennon ~/d_omp]$ gprof --line ./md.pg gmon.out  > report.gmon
[jonesm@lennon ~/d_omp]$ less report.gmon
Flat profile:
Each sample counts as 0.000976562 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
12.02      6.88     6.88                             dist (md.f90:302@40000000000065e0)
10.68     12.99     6.11                             dist (md.f90:300@40000000000065a0)
 9.71     18.55     5.56                             dist (md.f90:302@4000000000006bf1)
 8.95     23.67     5.12                             compute (md.f90:194@4000000000004fa0)
 7.50     27.96     4.29                             compute (md.f90:168@4000000000004e31)
 7.35     32.16     4.20                             compute (md.f90:167@4000000000004a80)
 6.67     35.98     3.82                             compute (md.f90:167@40000000000048f0)
 5.02     38.85     2.87 249749500   11.50    11.50  dist_ (md.f90:266@4000000000005d40)
 2.83     40.46     1.62                             dist (md.f90:305@4000000000006be1)
 2.06     41.64     1.18                             compute (md.f90:167@40000000000048e1)
 2.06     42.82     1.18                             compute (md.f90:188@40000000000048e2)
 2.02     43.97     1.15                             dist (md.f90:300@4000000000005e52)
 1.94     45.08     1.11                             compute (md.f90:194@4000000000004f51)
 1.45     45.91     0.83                             compute (md.f90:192@4000000000004b91)
 1.33     46.67     0.76                             dist (md.f90:305@4000000000006d01)
```

... and now let us take a look at the critical code sections,

```
! This potential is a harmonic well which smoothly saturates to a
! maximum value at PI/2.
!
  v(x) = ( sin ( min ( x, PI2 ) ) )**2
  dv(x) = 2.0D+00 * sin ( min ( x, PI2 ) ) * cos ( min ( x, PI2 ) )

  pot = 0.0D+00
  kin = 0.0D+00
```

which are implicit function declarations - the time consumption actually comes from where they are used, and not too suprisingly, it is the loop over particles that updates forces and momenta that is responsible for most of the consumed time:

```fortran
  do i = 1, np
!
!  Compute the potential energy and forces.
!
   f(1:nd,i) = 0.0D+00

  do j = 1, np

     if ( i /= j ) then

        call dist ( nd, pos(1,i), pos(1,j), rij, d )
!
!  Attribute half of the potential energy to particle J.
!
        pot = pot + 0.5D+00 * v(d)

        f(1:nd,i) = f(1:nd,i) - rij(1:nd) * dv(d) / d
```

Adding OpenMP directives to this loop:

```
!$OMP parallel do &
!$OMP default ( shared ) &
!$OMP shared ( nd ) &
!$OMP private ( i, j, rij, d ) &
!$OMP reduction ( + : pot, kin )
  do i = 1, np
!
!  Compute the potential energy and forces.
!
    f(1:nd,i) = 0.0D+00

    do j = 1, np

      if ( i /= j ) then

        call dist ( nd, pos(1,i), pos(1,j), rij, d )
!
!  Attribute half of the potential energy to particle J.
!
        pot = pot + 0.5D+00 * v(d)

        f(1:nd,i) = f(1:nd,i) - rij(1:nd) * dv(d) / d
```

so, based on these OpenMP directives, what kind of speedup can we get?

```
[jonesm@lennon ~/d_omp]$ module load intel
[jonesm@lennon ~/d_omp]$ ifort -O3 -o md.no-omp md.f90
[jonesm@lennon ~/d_omp]$ ifort -O3 -openmp -openmp_report2 -o md md.f90
[jonesm@lennon ~/d_omp]$ /usr/bin/time ./md.no-omp
November  5 2005   3:58:53.408 PM
MD
  A molecular dynamics program.
   100    124395.      0.226163      0.162282E-05
   200    124395.      0.918574      0.659101E-05
   300    124395.       2.07756      0.149066E-04
   400    124395.       3.70360      0.265724E-04
   500    124395.       5.79733      0.415922E-04
   600    124395.       8.35961      0.599709E-04
   700    124394.       11.3914      0.817147E-04
   800    124394.       14.8940      0.106831E-03
   900    124393.       18.8688      0.135327E-03
  1000    124393.       23.3172      0.167213E-03
MD
  Normal end of execution.
November  5 2005   3:59:49.310 PM
55.86user 0.00system 0:55.90elapsed 99%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (114major+43minor)pagefaults 0swaps
```

```
[jonesm@lennon ~/d_omp]$ setenv OMP_NUM_THREADS 2
[jonesm@lennon ~/d_omp]$ /usr/bin/time ./md
November  5 2005   4:00:31.129 PM
  MD
  A molecular dynamics program.
  The number of threads is    1
  This is processor    0
  This is processor    1
   100    124395.        0.226163        0.162282E-05
   200    124395.        0.918574        0.659101E-05
   300    124395.         2.07756        0.149066E-04
   400    124395.         3.70360        0.265724E-04
   500    124395.         5.79733        0.415922E-04
   600    124395.         8.35961        0.599709E-04
   700    124394.         11.3914        0.817147E-04
   800    124394.         14.8940        0.106831E-03
   900    124393.         18.8688        0.135327E-03
  1000    124393.         23.3172        0.167213E-03
  MD
  Normal end of execution.
November  5 2005   4:01:00.928 PM
59.44user 0.00system 0:29.86elapsed 199%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (155major+75minor)pagefaults 0swaps
```

```
[jonesm@lennon ~/d_omp]$ setenv OMP_NUM_THREADS 4
[jonesm@lennon ~/d_omp]$ /usr/bin/time ./md
November  5 2005   4:01:23.317 PM
  MD
  A molecular dynamics program.
  The number of threads is    1
   This is processor   0
   This is processor   1
   This is processor   2
   This is processor   3
    100    124395.      0.226163      0.162282E-05
    200    124395.      0.918574      0.659101E-05
    300    124395.       2.07756      0.149066E-04
    400    124395.       3.70360      0.265724E-04
    500    124395.       5.79733      0.415922E-04
    600    124395.       8.35961      0.599709E-04
    700    124394.       11.3914      0.817147E-04
    800    124394.       14.8940      0.106831E-03
    900    124393.       18.8688      0.135327E-03
   1000    124393.       23.3172      0.167213E-03
  MD
  Normal end of execution.
November  5 2005   4:01:38.260 PM
59.64user 0.00system 0:14.98elapsed 398%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (155major+81minor)pagefaults 0swaps
```