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, 2014

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.

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.

Synchronization Costs (cont'd)

- Overhead for synchronization on a 32-core Intel "Westmere" 2130MHz system (4 sockets, 8 cores/socket, Xeon E7-4530)

Nthreads	PARALLEL[μs]	DO[μs]	ATOMIC[μs]	REDUCTION[μs]
1	0.1	0.2	0.02	0.2
2	2.2	2.3	0.04	2.7
4	2.9	3.1	0.07	4.2
8	3.9	3.9	0.07	6.8
16	4.8	5.2	0.07	12.2
32	15.4	6.9	0.07	24.9

- 25μs$\times$2130MHz =**53250 clock cycles** - lost computation.
- Not exactly great progress ...

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?

```
1        real tmp,x
2  !$OMP PARALLEL DO REDUCTION(+:x)
3        do i=1,10000
4           tmp=dosomework(i)
5           x=x+tmp
6        end do
7  !$OMP END DO
8        y(iam) = work(x,iam)
9  !$OMP END PARALLEL
```


Race Conditions

- What is wrong with this code fragment?

```
1        real tmp,x
2  !$OMP PARALLEL DO REDUCTION(+:x)
3        do i=1,10000
4           tmp=dosomework(i)
5           x=x+tmp
6        end do
7  !$OMP END DO
8        y(iam) = work(x,iam)
9  !$OMP END PARALLEL
```

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

Race Conditions

- What about now?

```
1        real tmp,x
2  !$OMP PARALLEL DO REDUCTION(+:x),PRIVATE(tmp)
3        do i=1,10000
4           tmp=dosomework(i)
5           x=x+tmp
6        end do
7  !$OMP END DO NOWAIT
8        y(iam) = work(x,iam)
9  !$OMP END PARALLEL
```

Race Conditions

- What about now?

```
1       real tmp,x
2 !$OMP PARALLEL DO REDUCTION(+:x),PRIVATE(tmp)
3       do i=1,10000
4          tmp=dosomework(i)
5          x=x+tmp
6       end do
7 !$OMP END DO NOWAIT
8       y(iam) = work(x,iam)
9 !$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!

```
1       call OMP_INIT_LOCK(lock0)
2 !$OMP PARALLEL SECTIONS
3 !$OMP SECTION
4       call OMP_SET_LOCK(lock0)
5       iret = dolotsofwork()
6       if (iret.le.tol) then
7          call OMP_UNSET_LOCK(lock0)
8       else
9          call error(iret)
10      endif
11 !$OMP SECTION
12      call OMP_SET_LOCK(lock0)
13      call compute(A,B,iret)
14      call OMP_UNSET_LOCK(lock0)
15 $!OMP END SECTIONS
```


Load Balancing

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

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


Load Balancing

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

```
1  !$OMP PARALLEL SHARED (num_threads)
2  !$OMP SINGLE
3        num_threads = OMP_GET_NUM_THREADS()
4  !$OMP END SINGLE NOWAIT
5  !$OMP END PARALLEL
6  !$OMP PARALLEL DO PRIVATE(i,j,k)
7        do k = 1, num_threads
8           do i = k, n, num_threads
9              do j = 1,i
10                a(j,i) = a(j,i) + b(j)*c(j)
11             end do
12          end do
13       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)

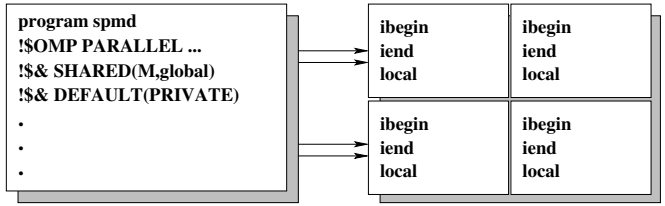
Domain Decomposition

- 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

```
1       program spmd
2  $!OMP PARALLEL DEFAULT(PRIVATE) SHARED(N,global)
3       num_threads = OMP_GET_NUM_THREADS()
4       iam = OMP_GET_THREAD_NUM()
5       ichunk = N/num_threads
6       ibegin = iam*ichunk
7       iend = ibegin + ichunk - 1
8       call lotsofwork(ibegin,iend,local)
9  $!OMP ATOMIC
10      global = global + local
11 $!OMP END PARALLEL
12      print*, global
13      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).

Thread Safety Issues

Certainly one must be careful about hidden state issues when calling functions/routines from multiple threads:

- MPI - check your level of thread safety with `MPI_Init_thread` and program accordingly
- Other functions - Up to you to check and ensure thread-safe functions (danger in treating any function as a black box)

Thread-safe Example

From the `rand` man page (section 3, RHEL 5):

```
#include <stdlib.h>

int rand(void);

int rand_r(unsigned int *seedp);

void srand(unsigned int seed);

DESCRIPTION
  The rand() function returns a pseudo–random integer between 0 and RAND_MAX.

  The srand() function sets its argument as the seed for a new sequence of
  pseudo–random integers to be returned by rand(). These sequences are repeatable
  by calling srand() with the same seed value.

  If no seed value is provided, the rand() function is automatically seeded with a value
  of 1.

  The function rand() is not reentrant or thread–safe, since it uses hidden state that is
  modified on each call. This might just be the seed value to be used by the next call,
  or it might be something more elaborate. In order to get reproducible behaviour in a
  threaded application, this state must be made explicit. The function rand_r() is
  supplied with a pointer to an unsigned int, to be used as state. This is a very small
  amount of state, so this function will be a weak pseudo–random generator. Try
  drand48_r(3) instead.
```


Lack of Max/Min in C/C++

Due to a lack of an intrinsic max/min function in C/C++, we have no built-in reduction operator in OpenMP, so one way to do so is to have each thread track its max/min value, and then update the global max/min accordingly with a protective directive:

```
1  #pragma omp parallel private(my_amax)
2  {
3    amax = 0;
4    my_amax = 0;
5    /* use private variable for max per thread */
6  #pragma omp for
7    for (i=0;i<=N;i++) {
8      if (a[i] > my_amax) {
9        my_amax = a[i];
10     }
11   } /* global update, requires only num_threads critical evaluations */
12 #pragma omp critical
13   if (my_amax > amax) {
14     {
15       amax = my_amax;
16     }
17   }
18 }
```


Max/Min with Locks

Another way to do max/min, this time with OpenMP locks:

```
1  omp_lock_t MAXLOCK;
2
3  omp_init_lock(&MAXLOCK);
4
5  #pragma omp parallel for
6  for (i = 0; i<Numberofelements; i++) {
7    if (array[i] > cur_max) {
8      omp_set_lock(&MAXLOCK);
9      if (array[i] > cur_max) {
10       cur_max = array[i];
11     }
12     omp_unset_lock(&MAXLOCK);
13   }
14
15 /* Destroying The Lock */
16 omp_destroy_lock(&MAXLOCK);
```


Example - Compare Max/Min with Critical vs. Lock

Compare the two methods - find the max in a randomly seeded array of varying size, serially and using the OpenMP critical and lock method (note that the outcome should be pretty obvious based on the two coding examples, but you can tinker with them to make the distinction less clear).

Thread Affinity

There are many times in which you may wish or need to specify how your compute threads get mapped to the physical (do not confuse physical with logical here) CPU cores:

- Contention for cache memory
- Contention for network interfaces (especially when combined with message-passing)

GNU Options

The GNU compilers currently support an option for CPU affinity, as well as an option for adjusting the available stack space per thread:

GOMP_CPU_AFFINITY : space-separated or comma-separated list of CPUs, either single CPU numbers in any order, a range of CPUs (M-N) or a range with some stride (M-N:S). Note that cores are counted starting from 0. Note that this view of CPU core reflects that of the operating system, which in many cases is not the full picture of the underlying hardware topology.

GOMP_STACKSIZE : sets the default thread stack size in kilobytes.

Intel Options

The Intel compilers support a much richer set of utilities for controlling the placement of threads:

- `KMP_AFFINITY` is the environment variable used, although the Intel run-time will also respect the `GOMP_CPU_AFFINITY` variable at a lower level of precedence.
- Details can be found (and are frequently changed as the architecture evolves) in the compiler documentation, but the latest as of this writing can be found at:

  http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/2011Update/cpp/lin/optaps/common/optaps_openmp_thread_affinity.htm
- Best bet is to review the documentation for the compiler that you are trying to use.
- Extremely helpful when simultaneous multi-threading (also known as hyper-threading) is turned on.

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 a shared memory environment.
- 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 x86_64	Gnu[a] (g77/gcc/g++)	2.5(>4.1),3.0(>4.4),3.1(>4.7),4.0[b](≥ 4.9)	-fopenmp
	PGI (pgf90/pgcc/pgCC)	2.5,3.0(≥ 12.0)	-mp
	Intel (ifort/icc/icpc)	2.5,3.0(≥ 11.0),3.1(≥ 12.1),4.0(≥ 14.0)	-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

[b]Incomplete support for 4.0, especially offload

Simple OpenMP example

```
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
```


CCR - simple example

```
[jonesm@rush ~/d_omp]$ module load intel
[jonesm@rush ~/d_omp]$ ifort -O3 -o simple_ifort -openmp simple.f90
[jonesm@rush ~/d_omp]$ setenv OMP_NUM_THREADS 4
[jonesm@rush ~/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@rush ~/d_omp]$ module load pgi
[jonesm@rush ~/d_omp]$ pgf90 -O3 -mp -o simple_pgi simple.f90
[jonesm@rush ~/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
```

CCR - simple example

```
[rush:~/d_omp]$ gcc -fopenmp -o hello2 hello2.c
[rush:~/d_omp]$ export OMP_NUM_THREADS=1
[rush:~/d_omp]$ ./hello2
Hello World from thread 0
There are 1 threads
[rush:~/d_omp]$ export OMP_NUM_THREADS=4
[rush:~/d_omp]$ ./hello2
Hello World from thread 1
Hello World from thread 3
Hello World from thread 0
Hello World from thread 2
There are 4 threads
[rush:~/d_omp]$ ./simple_ifort
 myid =           1   nthreads           4
 myid =           2   nthreads           4
 Number of available processors:          32
 myid =           0   nthreads           4
 myid =           3   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 ...

Compile for quick profiling and run to generate run-time profile (this is on a 12-core Xeon "Westmere" node):

```
1  [k08n08a:~/d_omp]$ ifort -O3 -o md1500-pg.x -g -p md1500.f90
2  [k08n08a:~/d_omp]$ /usr/bin/time ./md1500-pg.x
3  September 19 2011    1:39:35.156 PM
4
5  MD
6    A molecular dynamics program.
7
8
9      100   0.112109E+07   0.893929       0.158956E-10
10     200   0.112109E+07   3.63376       -0.189220E-10
11     300   0.112108E+07   8.23009       -0.115307E-09
12     400   0.112107E+07   14.7004       -0.304663E-09
13     500   0.112107E+07   23.0692       -0.619601E-09
14     600   0.112106E+07   33.3684       -0.109145E-08
15     700   0.112104E+07   45.6372       -0.175113E-08
16     800   0.112103E+07   59.9221       -0.263625E-08
17     900   0.112101E+07   76.2778       -0.377839E-08
18    1000   0.112099E+07   94.7666       -0.521232E-08
19
20 MD
21   Normal end of execution.
22
23 September 19 2011    1:42:13.397 PM
24 158.21user 0.00system 2:38.25elapsed 99%CPU (0avgtext+0avgdata 8032maxresident)k
25 0inputs+480outputs (0major+539minor)pagefaults 0swaps
```


Simple analysis based on profile:

```
1  [k08n08a:~/d_omp]$  gprof --line ./md1500-pg.x gmon.out > report-line-md1500.txt
2  [k08n08a:~/d_omp]$ less report-line-md1500.txt
3  Flat profile:
4
5  Each sample counts as 0.01 seconds.
6    %   cumulative   self              self     total
7   time   seconds   seconds    calls  ns/call  ns/call  name
8   44.51     62.31    62.31                             __libm_sse2_sincos
9   18.27     87.89    25.58                             compute (md1500.f90:194 @ 4035b4)
10   6.18     96.54     8.65                             compute (md1500.f90:194 @ 40359f)
11   6.18    105.18     8.65                             compute (md1500.f90:168 @ 4035a9)
12   3.63    110.26     5.08 2250748500     2.25     2.25  dist_ (md1500.f90:266 @ 403680)
13   3.20    114.74     4.48                             compute (md1500.f90:167 @ 40355d)
14   3.11    119.10     4.36                             compute (md1500.f90:167 @ 403544)
15   2.55    122.66     3.57                             compute (md1500.f90:192 @ 403561)
16   2.06    125.55     2.89                             compute (md1500.f90:192 @ 403551)
17   1.66    127.87     2.32                             compute (md1500.f90:194 @ 403569)
18   1.39    129.82     1.95                             compute (md1500.f90:188 @ 403521)
19   1.10    131.37     1.55                             dist (md1500.f90:300 @ 4036c1)
```

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

```
164 ! This potential is a harmonic well which smoothly saturates to a
165 ! maximum value at PI/2.
166 !
167   v(x) = ( sin ( min ( x, PI2 ) ) )**2
168   dv(x) = 2.0D+00 * sin ( min ( x, PI2 ) ) * cos ( min ( x, PI2 ) )
169
170   pot = 0.0D+00
171   kin = 0.0D+00
```


and not too surprisingly, it is the loop over particles that updates forces and momenta that is responsible for most of the consumed time:

```
178   do i = 1, np
179 !
180 !  Compute the potential energy and forces.
181 !
182   f(1:nd,i) = 0.0D+00
183
184   do j = 1, np
185
186     if ( i /= j ) then
187
188       call dist ( nd, pos(1,i), pos(1,j), rij, d )
189 !
190 !  Attribute half of the potential energy to particle J.
191 !
192       pot = pot + 0.5D+00 * v(d)
193
194       f(1:nd,i) = f(1:nd,i) - rij(1:nd) * dv(d) / d
```


Adding OpenMP directives to this loop:

```
173 !$OMP parallel do &
174 !$OMP default ( shared ) &
175 !$OMP shared ( nd ) &
176 !$OMP private ( i, j, rij, d ) &
177 !$OMP reduction ( + : pot, kin )
178   do i = 1, np
179 !
180 !  Compute the potential energy and forces.
181 !
182   f(1:nd,i) = 0.0D+00
183
184   do j = 1, np
185
186     if ( i /= j ) then
187
188       call dist ( nd, pos(1,i), pos(1,j), rij, d )
189 !
190 !  Attribute half of the potential energy to particle J.
191 !
192       pot = pot + 0.5D+00 * v(d)
193
194       f(1:nd,i) = f(1:nd,i) - rij(1:nd) * dv(d) / d
```


Using these OpenMP directives, what kind of speedup can we get?

```
[k14n08b:~/d_omp]$ module load intel/11.1
[k14n08b:~/d_omp]$ ifort -O3 -o md1500.no-omp.x md1500.f90
[k14n08b:~/d_omp]$ ifort -O3 -openmp -openmp_report2 -o md1500.omp.x md1500.f90
md1500.f90(65): (col. 7) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
md1500.f90(94): (col. 10) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
md1500.f90(173): (col. 7) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
md1500.f90(357): (col. 7) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
[k14n08b:~/d_omp]$ /usr/bin/time ./md1500.no-omp.x
September 19 2011   3:34:34.561 PM
 MD
  A molecular dynamics program.


   100  0.112109E+07  0.893929       0.158956E-10
   200  0.112109E+07   3.63376      -0.189220E-10
   300  0.112108E+07   8.23009      -0.115307E-09
   400  0.112107E+07   14.7004      -0.304663E-09
   500  0.112107E+07   23.0692      -0.619601E-09
   600  0.112106E+07   33.3684      -0.109145E-08
   700  0.112104E+07   45.6372      -0.175113E-08
   800  0.112103E+07   59.9221      -0.263625E-08
   900  0.112101E+07   76.2778      -0.377839E-08
  1000  0.112099E+07   94.7666      -0.521232E-08

MD
  Normal end of execution.
 September 19 2011   3:37:05.843 PM
151.26user 0.00system 2:31.28elapsed 99%CPU (0avgtext+0avgdata 4688maxresident)k
0inputs+0outputs (0major+329minor)pagefaults 0swaps
```

```
[k14n08b:~/d_omp]$ export OMP_NUM_THREADS=1
[k14n08b:~/d_omp]$ /usr/bin/time ./md1500.omp.x
September 19 2011   4:38:48.788 PM
MD
  A molecular dynamics program.
  This is thread    0 of    1
   100   0.112109E+07   0.893929       0.158956E-10
   200   0.112109E+07    3.63376      -0.189222E-10
   300   0.112108E+07    8.23009      -0.115307E-09
   400   0.112107E+07    14.7004      -0.304663E-09
   500   0.112107E+07    23.0692      -0.619601E-09
   600   0.112106E+07    33.3684      -0.109145E-08
   700   0.112104E+07    45.6372      -0.175113E-08
   800   0.112103E+07    59.9221      -0.263625E-08
   900   0.112101E+07    76.2778      -0.377839E-08
  1000   0.112099E+07    94.7666      -0.521232E-08

MD
  Normal end of execution.

September 19 2011   4:41:41.716 PM
172.90user 0.00system 2:53.12elapsed 99%CPU (0avgtext+0avgdata 7632maxresident)k
0inputs+0outputs (0major+519minor)pagefaults 0swaps
```

So the OpenMP overhead is reflected in $S(1) \simeq 0.87$.


```
[k14n08b:~/d_omp]$ export OMP_NUM_THREADS=2
[k14n08b:~/d_omp]$ /usr/bin/time ./md1500.omp.x
September 19 2011   4:22:46.538 PM
MD
  A molecular dynamics program.
  This is thread    0 of    2
  This is thread    1 of    2
   100   0.112109E+07   0.893929       0.158883E-10
   200   0.112109E+07    3.63376      -0.189195E-10
   300   0.112108E+07    8.23009      -0.115309E-09
   400   0.112107E+07    14.7004      -0.304673E-09
   500   0.112107E+07    23.0692      -0.619611E-09
   600   0.112106E+07    33.3684      -0.109145E-08
   700   0.112104E+07    45.6372      -0.175112E-08
   800   0.112103E+07    59.9221      -0.263624E-08
   900   0.112101E+07    76.2778      -0.377839E-08
  1000   0.112099E+07    94.7666      -0.521232E-08

MD
  Normal end of execution.

September 19 2011   4:24:14.577 PM
175.85user 0.03system 1:28.06elapsed 199%CPU (0avgtext+0avgdata 7920maxresident)k
0inputs+0outputs (0major+539minor)pagefaults 0swaps
```

For 2 threads, we are up to $S(2) \simeq 1.7$.


```
[k14n08b:~/d_omp]$ export OMP_NUM_THREADS=4
[k14n08b:~/d_omp]$ /usr/bin/time ./md1500.omp.x
September 19 2011   4:31:19.409 PM
MD
  A molecular dynamics program.
  This is thread    0 of    4
  This is thread    3 of    4
  This is thread    2 of    4
  This is thread    1 of    4
   100   0.112109E+07   0.893929       0.158875E-10
   200   0.112109E+07    3.63376      -0.189224E-10
   300   0.112108E+07    8.23009      -0.115307E-09
   400   0.112107E+07    14.7004      -0.304674E-09
   500   0.112107E+07    23.0692      -0.619610E-09
   600   0.112106E+07    33.3684      -0.109145E-08
   700   0.112104E+07    45.6372      -0.175112E-08
   800   0.112103E+07    59.9221      -0.263624E-08
   900   0.112101E+07    76.2778      -0.377839E-08
  1000   0.112099E+07    94.7666      -0.521232E-08

MD
  Normal end of execution.

September 19 2011   4:32:03.336 PM
175.37user 0.06system 0:44.03elapsed 398%CPU (0avgtext+0avgdata 8176maxresident)k
0inputs+0outputs (0major+560minor)pagefaults 0swaps
```

For 4 threads, we are up to $S(4) \simeq 3.4$.


```
[k14n08b:~/d_omp]$ export OMP_NUM_THREADS=8
[k14n08b:~/d_omp]$ /usr/bin/time ./md1500.omp.x
September 19 2011   4:33:16.000 PM
MD
  A molecular dynamics program.
  This is thread    0 of    8
  This is thread    4 of    8
  This is thread    6 of    8
  This is thread    5 of    8
  This is thread    7 of    8
  This is thread    3 of    8
  This is thread    2 of    8
  This is thread    1 of    8
   100   0.112109E+07   0.893929       0.158856E-10
   200   0.112109E+07    3.63376      -0.189249E-10
   300   0.112108E+07    8.23009      -0.115308E-09
   400   0.112107E+07    14.7004      -0.304673E-09
   500   0.112107E+07    23.0692      -0.619612E-09
   600   0.112106E+07    33.3684      -0.109145E-08
   700   0.112104E+07    45.6372      -0.175113E-08
   800   0.112103E+07    59.9221      -0.263624E-08
   900   0.112101E+07    76.2778      -0.377840E-08
  1000   0.112099E+07    94.7666      -0.521232E-08

MD
  Normal end of execution.

September 19 2011   4:33:37.909 PM
174.90user 0.04system 0:22.01elapsed 794%CPU (0avgtext+0avgdata 8768maxresident)k
0inputs+0outputs (0major+606minor)pagefaults 0swaps
```

For 8 threads, we are up to $S(8) \simeq 6.9$.

```
[k14n08b:~/d_omp]$ export OMP_NUM_THREADS=12
[k14n08b:~/d_omp]$ /usr/bin/time ./md1500.omp.x
September 19 2011   4:34:02.638 PM
MD
  A molecular dynamics program.
  This is thread    0 of   12
  This is thread    4 of   12
...
  This is thread   11 of   12
  This is thread    3 of   12
  This is thread    2 of   12
  This is thread    1 of   12
   100   0.112109E+07   0.893929       0.158859E-10
   200   0.112109E+07    3.63376      -0.189249E-10
   300   0.112108E+07    8.23009      -0.115308E-09
   400   0.112107E+07    14.7004      -0.304674E-09
   500   0.112107E+07    23.0692      -0.619612E-09
   600   0.112106E+07    33.3684      -0.109145E-08
   700   0.112104E+07    45.6372      -0.175112E-08
   800   0.112103E+07    59.9221      -0.263624E-08
   900   0.112101E+07    76.2778      -0.377840E-08
  1000   0.112099E+07    94.7666      -0.521232E-08

MD
  Normal end of execution.

September 19 2011   4:34:17.169 PM
173.98user 0.00system 0:14.60elapsed 1191%CPU (0avgtext+0avgdata 17712maxresident)k
0inputs+0outputs (0major+680minor)pagefaults 0swaps
```

For 12 threads (this is a 12-core node), we are up to $S(12) \simeq 10.4$.