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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.0</td>
<td>2.3</td>
<td>0.1</td>
<td>2.1</td>
</tr>
<tr>
<td>2</td>
<td>8.4</td>
<td>7.8</td>
<td>0.4</td>
<td>11.0</td>
</tr>
<tr>
<td>4</td>
<td>11.6</td>
<td>6.8</td>
<td>1.5</td>
<td>20.7</td>
</tr>
<tr>
<td>8</td>
<td>28.0</td>
<td>14.1</td>
<td>3.1</td>
<td>31.0</td>
</tr>
</tbody>
</table>

10µs? Isn’t that pretty small?

10µs × 250MHz = 2500 clock cycles - lost computation.
Overhead for synchronization on an SGI Altix 3700 (Intel 1300MHz Itanium2 processors)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.3</td>
<td>0.3</td>
<td>0.1</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>2.3</td>
<td>2.1</td>
<td>0.4</td>
<td>2.6</td>
</tr>
<tr>
<td>4</td>
<td>5.9</td>
<td>4.7</td>
<td>0.4</td>
<td>9.6</td>
</tr>
<tr>
<td>8</td>
<td>6.6</td>
<td>6.8</td>
<td>0.5</td>
<td>24.1</td>
</tr>
<tr>
<td>16</td>
<td>10.3</td>
<td>10.7</td>
<td>0.6</td>
<td>60.7</td>
</tr>
<tr>
<td>32</td>
<td>19.2</td>
<td>19.3</td>
<td>0.7</td>
<td>132</td>
</tr>
<tr>
<td>64</td>
<td>41.8</td>
<td>40.9</td>
<td>0.7</td>
<td>316</td>
</tr>
</tbody>
</table>

10µs? Isn’t that pretty small?
10µs × 1300MHz = 13000 clock cycles - lost computation.
Not exactly great progress...
Synchroniza

Overhead for synchronization on an Intel “Clover
dual quad-core 1.866GHz Xeon processors"

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.2</td>
<td>0.2</td>
<td>0.02</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>1.6</td>
<td>1.7</td>
<td>0.08</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>2.3</td>
<td>2.4</td>
<td>0.14</td>
<td>3.1</td>
</tr>
<tr>
<td>8</td>
<td>3.8</td>
<td>3.9</td>
<td>0.52</td>
<td>5.8</td>
</tr>
</tbody>
</table>

5.8μs × 1866MHz = 10823 clock cycles - lost computation.

Overhead for synchronization on an Intel “Nehalem” (dual quad-core 2.8GHz Xeon processors)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.01</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>1.1</td>
<td>1.1</td>
<td>0.04</td>
<td>1.2</td>
</tr>
<tr>
<td>4</td>
<td>1.2</td>
<td>1.2</td>
<td>0.05</td>
<td>1.5</td>
</tr>
<tr>
<td>8</td>
<td>1.7</td>
<td>1.8</td>
<td>0.05</td>
<td>2.5</td>
</tr>
</tbody>
</table>

2.5μs × 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.
What is wrong with this code fragment?

```fortran
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?

```c
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?

```plaintext
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
$OMP END PARALLEL
```

Y (iam) = work(x,iam)
Race Conditions

What about now?

```plaintext
real tmp, x
!
$OMP PARALLEL DO REDUCTION(+:x),PRIVATE(tmp)
do  i=1,10000
tmp=dosomework(i)
x=x+tmp
end do
!
$OMP END PARALLEL
!
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!
A somewhat artificial example of deadlock - watch that resources are freed if you are using locks!

```c
OMP_INIT_LOCK(lock0)
!$OMP PARALLEL SECTIONS
!$OMP SECTION
   OMP_SET_LOCK(lock0)
   iret = dolotsofwork()
   if (iret.le.tol) then
      OMP_UNSET_LOCK(lock0)
   else
      error(iret)
   endif
!$OMP SECTION
   OMP_SET_LOCK(lock0)
   compute(A,B,iret)
   OMP_UNSET_LOCK(lock0)
!$OMP END SECTIONS
```
Consider the following code fragment - can you see why it not efficient to parallelize on the outer loop?

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

```c
!$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
end do
```
Another equivalent (and somewhat cleaner!) way,

```c
!$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
end do
```
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!
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)
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 Single Program Multiple Data model (SPMD).
Coarse Grain SPMD Example

```fortran
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
```

M. D. Jones, Ph.D. (CCR/UB)
Coarse Grain SPMD Example

```
program spmd
    !$OMP PARALLEL ...
    !$& SHARED(M,global)
    !$& DEFAULT(PRIVATE)
    ...
```

Diagram:
- `program spmd`:
  - `!$OMP PARALLEL ...`
  - `!$& SHARED(M,global)`
  - `!$& DEFAULT(PRIVATE)`
  - `...`

- `ibegin`
- `iend`
- `local`

- `ibegin`
- `iend`
- `local`

- `ibegin`
- `iend`
- `local`

- `ibegin`
- `iend`
- `local`
**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.
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 compliance:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Compiler</th>
<th>OMP</th>
<th>Invocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux IA64</td>
<td>Gnu (g77/gcc/g++)</td>
<td>No</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Intel (ifort/icc/icpc)</td>
<td>2.5</td>
<td>-openmp -openmp_report2</td>
</tr>
<tr>
<td>Linux x86_64</td>
<td>Gnu(^a) (g77/gcc/g++)</td>
<td>2.5((&gt;4.1))</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>PGI (pgf90/pgcc/pgCC)</td>
<td>2.5</td>
<td>-mp</td>
</tr>
<tr>
<td></td>
<td>Intel (ifort/icc/icpc)</td>
<td>2.5,3.0((\geq 11.0))</td>
<td>-openmp -openmp_report2</td>
</tr>
</tbody>
</table>

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

---

M. D. Jones, Ph.D. (CCR/UB)  
Practical Issues in OpenMP  
HPC1 Fall 2009  
36 / 48
Practical OpenMP Example - Simple

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
```
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
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 ...
Practical OpenMP

Example - Molecular Dynamics

[jonesm@lennon ~]$ ifort -O3 -o md.pg -g -p md.f90
[jonesm@lennon ~]$ /usr/bin/time ./md.pg

November 5 2005 3:39:39.245 PM

MD
A molecular dynamics program.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>124395.</td>
<td>0.226163</td>
</tr>
<tr>
<td>200</td>
<td>124395.</td>
<td>0.918574</td>
</tr>
<tr>
<td>300</td>
<td>124395.</td>
<td>2.07756</td>
</tr>
<tr>
<td>400</td>
<td>124395.</td>
<td>3.70360</td>
</tr>
<tr>
<td>500</td>
<td>124395.</td>
<td>5.79733</td>
</tr>
<tr>
<td>600</td>
<td>124395.</td>
<td>8.35961</td>
</tr>
<tr>
<td>700</td>
<td>124394.</td>
<td>11.3914</td>
</tr>
<tr>
<td>800</td>
<td>124394.</td>
<td>14.8940</td>
</tr>
<tr>
<td>900</td>
<td>124393.</td>
<td>18.8688</td>
</tr>
<tr>
<td>1000</td>
<td>124393.</td>
<td>23.3172</td>
</tr>
</tbody>
</table>

MD
Normal end of execution.

November 5 2005 3:40:50.247 PM

70.94 user 0.00 system 1:11.23 elapsed 99% CPU (0 avgtext+0 avgdata 0 maxresident)k
0 inputs+0 outputs (116 major+120 minor) page faults 0 swaps
```
[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.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>seconds</th>
<th>self time</th>
<th>seconds</th>
<th>total time</th>
<th>seconds</th>
<th>self calls</th>
<th>ns/call</th>
<th>total calls</th>
<th>ns/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.02</td>
<td>6.88</td>
<td>6.88</td>
<td>dist (md.f90:302@40000000000065e0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.68</td>
<td>12.99</td>
<td>6.11</td>
<td>dist (md.f90:300@40000000000065a0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.71</td>
<td>18.55</td>
<td>5.56</td>
<td>dist (md.f90:302@4000000000006bf1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.95</td>
<td>23.67</td>
<td>5.12</td>
<td>compute (md.f90:194@4000000000004fa0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.50</td>
<td>27.96</td>
<td>4.29</td>
<td>compute (md.f90:168@4000000000004e31)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.35</td>
<td>32.16</td>
<td>4.20</td>
<td>compute (md.f90:167@4000000000004a80)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.67</td>
<td>35.98</td>
<td>3.82</td>
<td>compute (md.f90:167@4000000000004bf0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.02</td>
<td>38.85</td>
<td>2.87</td>
<td>dist_ (md.f90:266@4000000000005d40)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.83</td>
<td>40.46</td>
<td>1.62</td>
<td>dist (md.f90:305@4000000000006be1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.06</td>
<td>41.64</td>
<td>1.18</td>
<td>compute (md.f90:167@40000000000048e1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.06</td>
<td>42.82</td>
<td>1.18</td>
<td>compute (md.f90:188@40000000000048e2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.02</td>
<td>43.97</td>
<td>1.15</td>
<td>dist (md.f90:300@4000000000005e52)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.94</td>
<td>45.08</td>
<td>1.11</td>
<td>compute (md.f90:194@4000000000004f51)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.45</td>
<td>45.91</td>
<td>0.83</td>
<td>compute (md.f90:192@4000000000004b91)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.33</td>
<td>46.67</td>
<td>0.76</td>
<td>dist (md.f90:305@4000000000006d01)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
... and now let us take a look at the critical code sections,

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

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

```fortran
! 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)
!
$\mathbf{f}(1:nd,i) = \mathbf{f}(1:nd,i) - \mathbf{rij}(1:nd) \times \mathbf{dv(d)} / \mathbf{d}$
```
Adding OpenMP directives to this loop:

```c
!$OMP parallel do &
!$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)
!
   end if
!
   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.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>124395.</td>
<td>0.222163</td>
</tr>
<tr>
<td>200</td>
<td>124395.</td>
<td>0.918574</td>
</tr>
<tr>
<td>300</td>
<td>124395.</td>
<td>2.07756</td>
</tr>
<tr>
<td>400</td>
<td>124395.</td>
<td>3.70360</td>
</tr>
<tr>
<td>500</td>
<td>124395.</td>
<td>5.79733</td>
</tr>
<tr>
<td>600</td>
<td>124395.</td>
<td>8.35961</td>
</tr>
<tr>
<td>700</td>
<td>124394.</td>
<td>11.3914</td>
</tr>
<tr>
<td>800</td>
<td>124394.</td>
<td>14.8940</td>
</tr>
<tr>
<td>900</td>
<td>124393.</td>
<td>18.8688</td>
</tr>
<tr>
<td>1000</td>
<td>124393.</td>
<td>23.3172</td>
</tr>
</tbody>
</table>

MD

Normal end of execution.

November 5 2005 3:59:49.310 PM

55.86 user 0.00 system 0:55.90 elapsed 99%CPU (0 avgtext+0 avgdata 0 maxresident)k
0 inputs+0 outputs (114 major+43 minor) pagefaults 0 swaps
```
[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
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>124395.</td>
<td>0.226163</td>
<td>0.162282E−05</td>
</tr>
<tr>
<td>200</td>
<td>124395.</td>
<td>0.918574</td>
<td>0.659101E−05</td>
</tr>
<tr>
<td>300</td>
<td>124395.</td>
<td>2.07756</td>
<td>0.149066E−04</td>
</tr>
<tr>
<td>400</td>
<td>124395.</td>
<td>3.70360</td>
<td>0.265724E−04</td>
</tr>
<tr>
<td>500</td>
<td>124395.</td>
<td>5.79733</td>
<td>0.415922E−04</td>
</tr>
<tr>
<td>600</td>
<td>124395.</td>
<td>8.35961</td>
<td>0.599709E−04</td>
</tr>
<tr>
<td>700</td>
<td>124394.</td>
<td>11.3914</td>
<td>0.817147E−04</td>
</tr>
<tr>
<td>800</td>
<td>124394.</td>
<td>14.8940</td>
<td>0.106831E−03</td>
</tr>
<tr>
<td>900</td>
<td>124393.</td>
<td>18.8688</td>
<td>0.135327E−03</td>
</tr>
<tr>
<td>1000</td>
<td>124393.</td>
<td>23.3172</td>
<td>0.167213E−03</td>
</tr>
</tbody>
</table>

MD
Normal end of execution.
November  5 2005  4:01:00.928 PM
59.44 user 0.00 system 0:29.86 elapsed 199%CPU (0 avgtext+0 avgdata 0 maxresident)k
0 inputs+0 outputs (155 major+75 minor) pagefaults 0 swaps
Practical OpenMP

Example - Molecular Dynamics

[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

59.64 user  0.00 system  0:14.98 elapsed 398%CPU (0 avgtext+0 avgdata 0 maxresident)k
0 inputs+0 outputs (155 major+81 minor) page faults  0 swaps