OpenMP Programming: Correctness, Tuning, Examples

Reinhold Bader (LRZ)
Georg Hager (RRZE)
Some pitfalls using OpenMP

- **OpenMP is easier to handle**
  - no explicit mapping of data to processors necessary

- **But: possible resource conflicts**
  - incorrectly programmed access to shared resource

- **Three major classes of errors:**
  - race condition: results of program depend on detailed timing of threads → usually incorrect
  - deadlock: (some) threads lock up waiting for a resource which is never freed
  - livelock: multiple threads work forever on individual tasks
OpenMP errors: race condition example

```c
%!omp parallel sections
%!omp section
a = b + c
%!omp section
b = a + c
%!omp section
c = b + a
%!omp end parallel sections

may produce wrong answers
varying from run to run and with number of threads

Use suitable synchronization construct to fix
may effectively serialize program!

ic = 0
%!omp parallel sections
%!omp section
a = b + c
ic = 1
%!omp flush(ic)
%!omp section
    do while (ic < 1)
%!omp flush(ic)
    end do
b = a + c
ic = 2
... (etc)
%!omp end parallel sections
```
OpenMP errors: deadlock example

call omp_init_lock(var)
!$omp parallel sections
!$omp section
    call omp_set_lock(var)
    ... (work)
    if (ival > tol) then
        call omp_unset_lock(var)
    else
        call error(ival)
    end if
!$omp section
    call omp_set_lock(var)
    ... (work2 depends on work)
    call omp_unset_lock(var)
!$omp end parallel sections

var potentially never unlocked
OpenMP errors: livelock example

```
!$omp parallel private(myid)
  myid = omp_get_thread_num()
  n = omp_get_num_threads()
  do while (res*res > tol)
    p(myid) = update(ii,myid)
    !$omp single
    res = match(p, n)
    !$omp end single
  end do
!$omp end parallel
```

Consequence:
do while exit condition may never be fulfilled even if serial program works correctly
Telling the Intel compiler and runtime what to do

**Compiler options**
- `-openmp`: activate OpenMP directives. Note that this implies `-auto` (all automatic variables go to the stack)
- `-openmp_stubs`: use stub library, produce standard sequential code
- `-openmp_report[0|1|2]`: produces further diagnostic information

**Runtime options**
- **stack size**: If you get segmentation faults, try `ulimit -s 1000000` (default is 8192 = 8 MByte)
- **OMP_*** variables
- Intel-specific settings:
  - `KMP_STACKSIZE`: sets the thread-private stack size to specified value, e.g. `KMP_STACKSIZE=2g` for 2 GByte. Default is 4m.
  - `KMP_SCHEDULE`: change the default schedule chunk size to yield better distribution of workload
Intel runtime (cont’d)

Intel-specific settings (cont’d)

- **KMP_LIBRARY**: OpenMP execution mode can be set to one of the following:
  - **throughput**: threads will be put to sleep after a waiting time determined by the `KMP_BLOCKTIME` env. variable. Useful if resources are shared. This is the default setting
  - **turnaround**: worker threads stay active all the time, do not yield for other threads. Useful if dedicated resources available

- **KMP_BLOCKTIME**: set to the waiting time for throughput mode. In units of milliseconds. Default is 200.
  - other units are possible, e.g. `KMP_BLOCKTIME=4s` specifies 4 seconds
- **KMP_ALL_THREADS**: maximum number of threads allowed in a parallel region. Default is
  - 4 * (number of CPUs)

**Further available**
- KMP library routines for performing settings and thread-local memory managent at run time.
Example (Performance Issue):
Counting even and odd elements in an array

```
integer ict(2,ntdm)
!$omp parallel private(myid)
  myid = omp_get_thread_num()+1
!$omp do private(index)
  do i=1,n
    index = mod(ia(i),2)+1
    ict(index,myid) = &
    & ict(index,myid) + 1
  end do
!$omp end do
!$omp critical
  is = is + ict(1:2,myid)
!$omp end critical
!$omp end parallel
```

<table>
<thead>
<tr>
<th>No. of Threads</th>
<th>4-way SMP</th>
<th>Altix</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>1,18</td>
<td>1,13</td>
</tr>
<tr>
<td>1</td>
<td>1,17</td>
<td>1,14</td>
</tr>
<tr>
<td>2</td>
<td>4,5</td>
<td>2,25</td>
</tr>
<tr>
<td>4</td>
<td>6,7</td>
<td>2,22</td>
</tr>
<tr>
<td>6</td>
<td>n/a</td>
<td>6,38</td>
</tr>
</tbody>
</table>
Performance Issue: false sharing

- When using shared data, multiple threads may
  - write to data adjacent in memory
  - make updates to same memory location (e.g. via critical)
  - in both cases, cache lines are (potentially) invalidated
  - result: increased interconnect traffic due to reload of cache lines
    - bad performance/limited scalability

- Therapy:
  - privatize, postpone updates to common memory location
  - increase problem size
  - increase chunk size
  - sometimes compiler optimization also helps
How to fix our example

integer ict(2)
!$omp parallel private(myid,ict)
myid = omp_get_thread_num()+1
ict = 0
!$omp do private(index)
do i=1,n
    index = mod(ia(i),2)+1
    ict(index) = &
    & ict(index) + 1
end do
!$omp end do
!$omp critical
    is = is + ict
!$omp end critical
!$omp end parallel

Execution Times
(seconds for 100M Elem.)

<table>
<thead>
<tr>
<th>No. of Threads</th>
<th>4-way SMP</th>
<th>Altix</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>1,17</td>
<td>1,13</td>
</tr>
<tr>
<td>1</td>
<td>1,10</td>
<td>1,05</td>
</tr>
<tr>
<td>2</td>
<td>0,55</td>
<td>0,53</td>
</tr>
<tr>
<td>4</td>
<td>0,28</td>
<td>0,27</td>
</tr>
<tr>
<td>6</td>
<td>n/a</td>
<td>0,18</td>
</tr>
</tbody>
</table>
Now one small change – insert updates to array

... (as before)

```fortran
!$omp do private(index)
   do i=1,n
      index = mod(ia(i),2)+1
      ict(index) = ict(index) + 1
      ia(i) = ia(i) - ict(index)
   end do
!$omp end do
... (as before)
```

<table>
<thead>
<tr>
<th>No. of Threads</th>
<th>4-way SMP</th>
<th>Altix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0,46</td>
<td>0,39</td>
</tr>
<tr>
<td>2</td>
<td>0,24</td>
<td>0,22</td>
</tr>
<tr>
<td>4</td>
<td>0,17</td>
<td>0,14</td>
</tr>
</tbody>
</table>

For SMP:
- expected result
- one memory interface for 4 CPUs

For Altix:
- should scale better
- one memory interface used here for each Thread!
Requirement for ccNUMA programming – distributed first touch

Need to initialize the array using the same worksharing scheme as for workload loop

```c
!$omp parallel do
do i=1,n
    ia(i) = ...
end do
!$omp end parallel do
```

... (here comes the workload loop)

### Execution Times (seconds for 100M Elem.)

<table>
<thead>
<tr>
<th>No. of Threads</th>
<th>4-way SMP</th>
<th>Altix with F.T.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0,46</td>
<td>0,39</td>
</tr>
<tr>
<td>2</td>
<td>0,24</td>
<td>0,20</td>
</tr>
<tr>
<td>4</td>
<td>0,17</td>
<td>0,10</td>
</tr>
</tbody>
</table>

For Altix:
- improved 2 and 4 thread results by 10% and 40%, respectively
- For **scalability** correctly distributed first touch is a **must**
Further remarks on first touch

- If only 2-way OpenMP is used (e.g. in hybrid parallel codes)
  - distributed first touch probably not needed if locality of threads to memory channel can be assured
  - this applies to Altix 3700 but not to Altix 4700 in Phase 1

- Iteratively working on tasks of varying sizes
  - need to use dynamic memory management
  - no “untouch” or “retouch” available (yet?)
  - deallocate $\rightarrow$ allocate $\rightarrow$ first touch $\rightarrow$ work loop $\rightarrow$ etc.

Without distr. first touch:
- physical location of memory concentrated on single nodes
- Congestion of NUMALink

With distr. first touch:
- memory evenly distributed
- mostly local memory traffic
Example:
First order linear recurrence

First order linear recurrences are of the following structure:

! with positive increment
\[ x(1)=... \]
\[
\text{do } i = 2,n \\
\quad x(i)=x(i)+b(i)*x(i-1) \\
\quad \text{or} \\
\quad x(i)=a(i)+b(i)*x(i-1)
\]
\[
\text{enddo}
\]

! with negative increment
\[ x(n)=... \]
\[
\text{do } i = n-1,1,-1 \\
\quad x(i)=x(i)+b(i)*x(i+1) \\
\quad \text{or} \\
\quad x(i)=a(i)+b(i)*x(i+1)
\]
\[
\text{enddo}
\]

\[
Bx = \begin{pmatrix}
1 & -b_2 & 1 \\
-1 & 1 & -b_3 \\
\vdots & \ddots & \ddots \\
& & -b_n & 1
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3 \\
\vdots \\
x_n
\end{pmatrix}
= \begin{pmatrix}
a_1 \\
a_2 \\
a_3 \\
\vdots \\
a_n
\end{pmatrix}
\]
Parallel Algorithm for Recursion: “Partitioning“

Step 1: Solve in parallel:

\[ B^{(k)} \ast y^{(k)} = a^{(k)} \]

Step 2: Correct for error:

\[ \delta x^{(k)} = x^{(k)} - y^{(k)} \]

\[ (B \ast \delta x)^{(k)} = \begin{bmatrix} b_{j(k-1)+1} y_{j(k-1)} & 0 & \cdots & 0 \end{bmatrix} \]

since only first element of rhs is different from zero only products of b occur
Code for the partitioning algorithm

\[
\text{nb} = \frac{n}{\text{num\_th}} \text{! if divisible}
\]

\[
!\text{omp parallel private(i,istart,iend,l)}
\]

\[
!\text{omp do}
\]

\[
\text{do } l=1,\text{num\_th}
\]

\[
\text{ istart} = (l-1)\text{\_}nb + 1
\]

\[
\text{ iend} = \text{min}(n, \text{ l\_nb})
\]

\[
\text{x(istart)} = a(\text{istart})
\]

! further optimization on \textbf{recurrence}
! possible; overwrite \text{x} and \text{b} to save
! space / memory operations

\[
\text{do } i=\text{istart+1},\text{iend}
\]

\[
\text{x}(i) = a(i) + b(i) \times x(i-1)
\]

\[
\text{b}(i) = b(i) \times b(i-1)
\]

\[
\text{end do}
\]

\[
\text{end do}
\]

\[
!\text{omp end do}
\]

! now comes the sequentialized part
! only last element in each block
! corrected here

\[
!\text{omp single}
\]

\[
\text{do } l=2,\text{num\_th}
\]

\[
\text{ istart} = (l-1)\text{\_}nb + 1
\]

\[
\text{ iend} = \text{min}(n, \text{ l\_nb})
\]

\[
\text{x(iend)} = x(\text{iend}) + b(\text{iend}) \times x(\text{istart}-1)
\]

\[
\text{end do}
\]

\[
!\text{omp end single}
\]

! and then do remaining updates in parallel

\[
!\text{omp do}
\]

\[
\text{do } l=2,\text{num\_th}
\]

\[
\text{ istart} = (l-1)\text{\_}nb + 1
\]

\[
\text{ iend} = \text{min}(n, \text{ l\_nb})
\]

! no dependencies here \rightarrow \textbf{vectorize}

\[
\text{do } i=\text{istart},\text{iend}-1
\]

\[
\text{x}(i) = x(i) + b(i) \times x(\text{istart}-1)
\]

\[
\text{end do}
\]

\[
\text{end do}
\]

\[
!\text{omp end do}
\]

\[
!\text{omp end parallel}
\]
Alternative: Parallelization by cyclic reduction

\[ x(i) = a(i) + b(i) \cdot x(i-1) \] is transformed to \[ x(i) = a'(i) + b'(i) \cdot x(i-nb) \]

where

\[ a'(i) = \sum a(i-v) \prod b(i-\mu) \quad b'(i) = \prod b(i-v) \]

which can be executed in parallel

\[ x(1), x(nb+1), x(2*nb+1), \ldots \] must be calculated sequentially

\[ \rightarrow \text{only as many elements as there are threads (-1)} \]

remaining recursions within blocks can then be done in parallel

\[ \rightarrow \text{Block } i \ (x(i*nb+1), x(i*nb+2), \ldots) \text{ by Thread } i \]
nb = n/num_th  ! if divisible
!$omp parallel private(i,istart,iend,l,aa,bb)
!$omp do
  do l=1,num_th
    istart = (l-1)*nb + 1
    iend = min(n, l*nb)
    if (l < num_th) then
      bb = b(iend+1)
      aa = a(iend+1)
      do i=iend,istart,-1
        aa = aa + bb * a(i)
        bb = bb * b(i)  ! b(1) must be 1.0
      end do
    end if
  end do
!$omp end do
! now comes the sequentialized part
! we solve the reduced system,
! obtaining only num_th components
! x(istart) of the solution
!$omp ordered
  do l=1,num_th
    istart = (l-1)*nb + 1
    iend = min(n, l*nb)
    if (l == 1) x(1) = a(1)
  end do
!$omp ordered
  ! and then do updates in parallel
  !$omp do
    do l=1,num_th
      istart = (l-1)*nb + 1
      iend = min(n, l*nb)
      do i=istart+1,iend
        x(i) = a(i) + b(i) * x(i-1)
      end do
    end do
  !$omp end do
  !$omp end parallel
Properties of the algorithms

- Both solutions require 5n operations instead of the 2n for the sequential algorithm
  - would expect optimal speedup of 2/5 \times \text{(number of threads)}

Memory operations

- partitioning: 5n loads, 3n stores
- cyclic reduction: 5n loads, n stores
- sequential: 3n loads, n stores

Cyclic reduction can also be used to vectorize/pipeline sequential code ("reductive" loop unrolling)
Comparing performance: Hitachi SR8000 and Altix Speedups

Recursion using 8 OpenMP threads

- SR8000 - cyclicred
- SR8000 - partition
- Altix - cyclicred_stretched
- Altix - cyclicred
- Altix - partition_stretched

Speedup vs. Vector Length
“Stretched and non-stretched” mode on Altix

- Memory bus is shared by 2 CPUs on each subnode
  - from memory, system scales only with every second CPU

- How to start up in stretched mode
  - explicit CPU placement
    ```
dplace -e -c 0,x,2,4,6,8,10,12,14 ./myprog.exe
    ```

- In contrast to normal mode
  ```
dplace -e -c 0,x,1-7 ./myprog.exe
    ```
Intel Threading Tools

Thread Checker

Thread Profiler
What do these tools provide?

- **Thread Checker**
  - monitors application during execution to locate **threading issues** (deadlocks, stalls, race conditions)
  - isolates bugs to source lines, suggests resolution
  - categorizes severity of bug

- **Thread Profiler**
  - monitors application during execution to identify **performance issues** (overhead, synchronization impact)
  - provides per-thread performance data and information about load-balance problems

- **OpenMP and pthreads are supported**
- **Both tools originate from Kuck&Ass.**
  - bought by Intel around 6 years ago.
Thread Checker: Usability for Itanium systems

GUI necessary for full analysis (binary instrumentation)
- but only available for Windows
  - integrated into VTune environment
- hence, must attach to Linux system and do remote data collection (RDC)
  - ports 50000, 50001, 50002 must be open → firewalls (ssh tunnel?)

Limitations on Itanium
- only thread count independent analysis supported
  - `omp_get_thread_num()` and other thread-count dependent functions cannot be checked
- essentially limited to stand-alone analysis via combined `-openmp -tcheck` compiler options
  - transfer result dataset to Windows system for analysis
Using Thread Checker on x86

- **this is fully supported**
  - build program using the `-openmp` switch (no `-tcheck`)
  - run via RDC with binary instrumentation performed by VTune

**Example here:**
- run code on Windows
  - options /optimize /Qopenmp /Qtcheck /debug
- generate new VTune project
  - use threading Wizard
  - specify location of executable
  - don’t use NFS or AFS
- Wizard runs activity
x = 1
z = 5
!

write (6, *) 'Threads: ', omp_get_num_threads() !

11 !

!

write (6, *) 'Threads: ', omp_get_num_threads() !

!

write (6, *) 'x = ', x
write (6, *) 'i = ', i
write (6, *) 'y(1) = ', y(1)
Thread Profiler

- **Windows version includes GUI**
  - delivered as part of VTune
  - also RDC from Linux machine
    - couldn’t get it to run

- **Example here:**
  - matrix-vector code run on Windows
    - options /optimize /Qopenmp-profile /debug
    - use 2 threads on single-CPU machine
  - generate new VTune project
    - use threading Wizard
    - specify location of executable
    - select OpenMP (not threading) mode and number of threads
  - Wizard runs activity
Remarks on profiler results

- need run with representative number of threads to properly judge
- drill-down to source
  - possible for each parallel region
- resolution of timeline legend
  - horrific
- see a lot of barrier for matrix-vector
  - is in effect an imbalance

suggeston on what to fix?
- still need to think about that yourself
  → more diagnostic than therapeutic