OpenMP on MPPs and clusters of SMP nodes using Intel® Compilers with Cluster OpenMP

Rolf Rabenseifner, Bettina Krammer
rabenseifner@hlrs.de, krammer@hlrs.de

University of Stuttgart
High-Performance Computing-Center Stuttgart (HLRS)
www.hlrs.de

Oct 17, 2006
Used version of Intel® Compiler with Cluster OpenMP under Linux: Compiler 9.1

Goal

- To run OpenMP parallel applications on clusters
- Ease of OpenMP parallelization on cheap clusters
- Instead of
  - expensive MPI parallelization, or
  - expensive shared memory / ccNUMA hardware

OpenMP only

Intel® Compilers with Cluster OpenMP – Consistency Protocol

Basic idea:

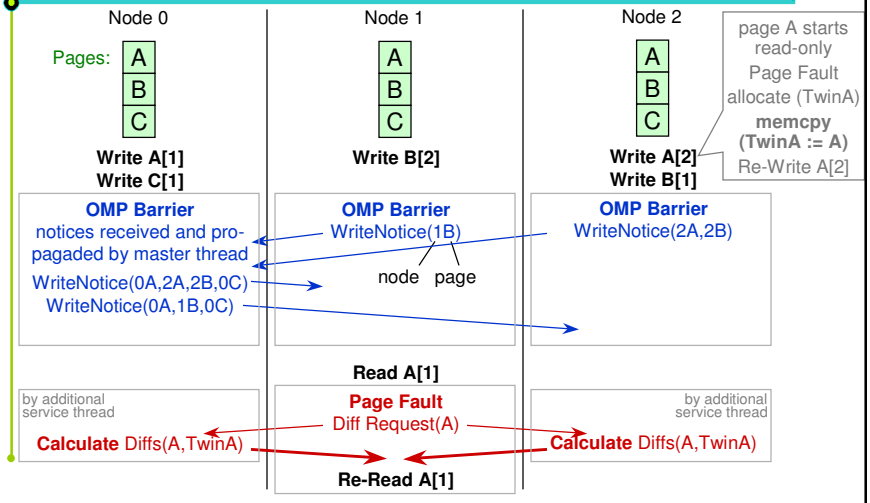
- Between OpenMP barriers, data exchange is not necessary, i.e., visibility of data modifications to other threads only after synchronization.
- When a page of sharable memory is not up-to-date, it becomes ***protected.***
- Any access then faults (SIGSEGV) into Cluster OpenMP runtime library, which requests info from remote nodes and updates the page.
- Protection is removed from page.
- Instruction causing the fault is re-started, this time successfully accessing the data.

Intel Cluster OpenMP [20a]

H L R S

Courtesy of J. Cownie, Intel

Consistency Protocol Detail of Intel® Cluster OpenMP



H L R S

Courtesy of J. Cownie, Intel

Real consistency protocol is more complicated

- Diffs are done only when requested
- Several diffs are locally stored and transferred later if a thread first reads a page after several barriers.
- Each write is internally handled as a read followed by a write.
- If too many diffs are stored, a node can force a "reposession" operation, i.e., the page is marked as invalid and fully re-sent if needed.
- Another key point:
  - After a page has been made read/write in a process, no more protocol traffic is generated by the process for that page until after the next synchronization (and similarly if only reads are done once the page is present for read).
  - This is key because it's how the large cost of the protocol is averaged over many accesses.
  - I.e., protocol overhead only "once" per barrier
- Examples in the Appendix

Courtesy of J. Cownie, Intel

hybrid MPI+OpenMP ↔ OpenMP only

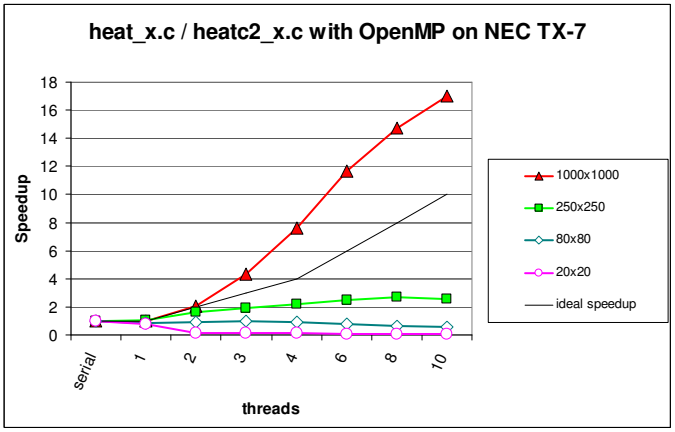
Comparison: MPI based parallelization ←→ DSM

- MPI based:
  - Potential of boundary exchange between two domains in one large message
    - → Dominated by **bandwidth** of the network
- DSM based (e.g. Intel® Cluster OpenMP):
  - Additional latency based overhead in each barrier
    - → May be marginal
  - Communication of **updated data of pages**
    - → Not all of this data may be needed
    - → i.e., too much data is transferred
    - → Packages may be too small
    - → Significant latency
  - Communication not oriented on boundaries of a domain decomposition
    - → probably more data must be transferred than necessary

by rule of thumb:
Communication may be 10 times slower than with MPI

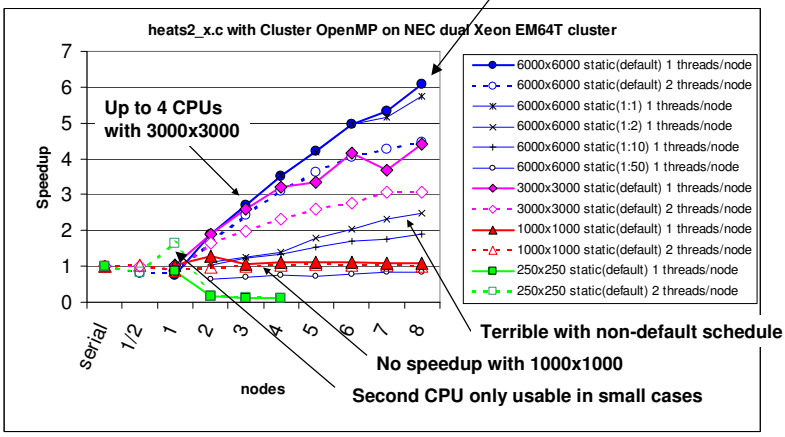
Comparing results with heat example

- Normal OpenMP on shared memory (ccNUMA) NEC TX-7



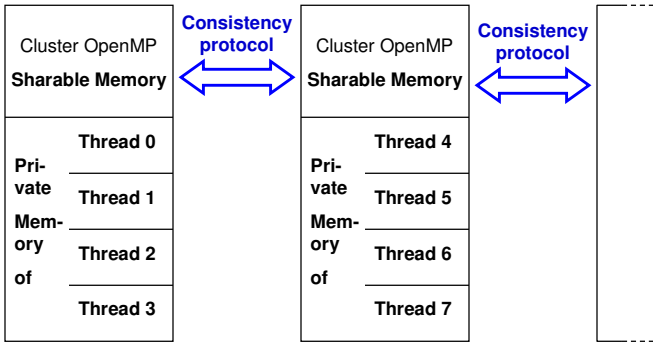
Heat example: Cluster OpenMP Efficiency

- Cluster OpenMP on a Dual-Xeon cluster



Key concept – Sharable Memory

- Shared variables must reside in **sharable memory**



- Usually:
  - one process per SMP node
  - one thread per CPU-core

Usage – Overview

- Variables that are used with "shared" data scope must be allocated as "**sharable**"
- If data declaration and "shared" data scope is within same lexicographic scope, **sharable** is done automatically
- Otherwise
  - C: #pragma intel omp sharable(var1, …)
  - Fortran: !dir$ omp sharable(var1, ….)
- malloc() and Fortran Cray pointer must be substituted by
  - kmp_sharable_malloc()
  - kmp_sharable_realloc()
  - kmp_sharable_ralloc()
  - kmp_sharable_free()
- Fortran call by reference may imply that a temporary variable must be used

```
!dir$ omp sharable(ntemp)
! call func(expression)
ntemp = expression
call func(ntemp)

subroutine func(n)
!omp parallel do shared(n)
do i=1,n
  ...
```

Porting – Step 1

- Try the program with Cluster OpenMP
  - If it works → done ➔ **see Example 1**
  - If not → go to Step 2

Porting Step 2: Using –clomp–sharable–propagation

- Compile all sources
  - `ifort -cluster-openmp` **`-clomp-sharable-propagation -ipo`** `file.f file2.f`
- At the "link" step, sharable directive warnings will indicate variables that must be made sharable
  - `fortcom: Warning: Sharable directive should be inserted by user as '!dir$ omp sharable(n)' in file file.f, line 23, column 16`

Porting Step 2: –clomp–sharable–propagation Example

```
#include <stdio.h>
void f(double *d)
{ int i;
#pragma omp parallel for \
                    shared(d) [1])
  for (i=0; i<10; i++)
    d[i] = 1./(i+1);
}

int main()
{ int i;
  double x[10];
#pragma intel omp sharable(x)
  f(x);
  for (i=0; i<10; i++)
    printf("%f\n",x[i]);
}
```

[1]) **Not needed because default**

Necessity detected by linker with –clomp–sharable–propagation –ipo

shared(d) clause is **outside** of lexicographical scope of the real memory declaration x[10]

Therefore,

sharable(x) is necessary to guarantee that x[10] is in the sharable memory!

CAUTION:
#pragma ... sharable(...) must be located **after** the declaration

H L R S

➔ **see Example 2 (and 3 Fortran)**

Porting Step 3a: Dynamic Sharable Allocation

- C API – just like malloc/free

```
#include <omp.h>
void *ptr;
ptr = kmp_sharable_malloc(size);
ptr = kmp_sharable_realloc(ptr, size);
ptr = kmp_sharable_calloc(n, size);
kmp_sharable_free(ptr);
```

- Fortran API

```
include 'omp_lib.h' ! or use omp_lib
        real, allocatable :: a(:)
!dir$ omp sharable(a)
        allocate(a(count))

        real a(100); pointer (p,a); p = kmp_sharable_malloc(400)
```


H L R S

➔ **see Example 3: f_exa3.f**

Porting Step 3b: Using KMP_DISJOINT_HEAPSIZE

If you didn't found all allocated memory that must be made sharable:

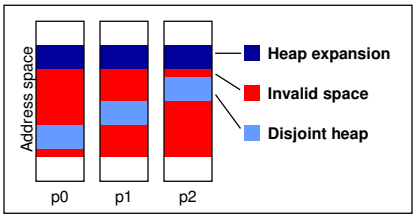
- Compile with "–g"
- export KMP_DISJOINT_HEAPSIZE=<size> (minimum: 2M) [sh, ksh, bash]
  setenv KMP_DISJOINT_HEAPSIZE <size> [csh, tcsh]
- Run program
- If you get an error:

  Cluster OMP Fatal: Proc#0 Thread#2 (INITIAL): Segmentation fault (ip=0x400c2f address=0x5a85c20)
  Cluster OMP Fatal: Proc#0 Thread#2 (INITIAL): This is an address in the local heap for process 1

- Use addr2line to find source line:

  % addr2line –e my_prog 0x400c2f /home/jhcownie/tmp/my_prog.c:17

- Check the malloc/allocate of data used on that line



Porting Step 3c: Example

```
#include <stdlib.h>
#include <stdio.h>
#define N 10

int main()
{ int i;
  double *x;
# ifdef _CLUSTER_OPENMP
   x = kmp_sharable_malloc(N*sizeof(double));
# else
   x = malloc(N*sizeof(double));
# endif
#pragma omp parallel for
  for (i=0; i<N; i++) x[i] = i*i;
  for (i=0; i<N; i++) printf("%d**2 = %f \n", i, x[i]);
}
```

Solution

Found by using KMP_DISJOINT_HEAPSIZE and addr2line

Porting Step 4:
Made Sharable with Compiler Option (Fortran)

Original code — use this option — makes it as if you wrote this code

```
common /blk/ a(100)
```

-clomp-sharable-commons →

```
common /blk/ a(100)
!dir$ omp sharable (/blk/)
```

➔ **Alternative approach for Example 3**

```
real a(100)
save a
```

-clomp-sharable-localsaves →

```
real a(100)
save a
!dir$ omp sharable (a)
```

```
module m
real a(100)
```

-clomp-sharable-modvars →

```
module m
real a(100)
!dir$ omp sharable (a)
```


Porting Step 5: (C++ only)
C++ Dynamic Sharable Allocation

- In all cases, use `#include <kmp_sharable.h>`
- To make all objects of certain class sharable
  - Convert `class foo { … };`
    to `class foo: public kmp_sharable_base { … };`
- To make instances of objects sharable
  - Convert `bah *p=new bah (…);`
    to `bah *p=new kmp_sharable bah(…);`
- To make STL containers and contents sharable
  - Convert `vector<int> *vp=new vector<int>;`
    to
    ```
    vector<int,kmp_sharable_allocator<int> >
    *vp=  new kmp_sharable
    vector<int,kmp_sharable_allocator<int> >;
    ```
    Space required

➔ **see Example 3: cpp_ex3.cpp**

Acknowledgements

- This lecture is based on an Intel Cluster OpenMP Tutorial (with examples 1-3) by Larry Meadows and James Cownie
- The Intel® Cluster OpenMP (compiler 9.1) was installed by Dmitri Chubarov and Bettina Krammer

Summary

- **Intel® Cluster OpenMP** can be used for programs with small communication foot-print!
- Source code modification needed: shared variables must be allocated in ***sharable*** memory
- It works!
- But efficiency strongly depends on type of application!

For the appropriate application a suitable tool!

Intel® Cluster OpenMP – Practical (on cacau.www.de)

Initialization

- **cd** ~/**OpenMP**/#nr/**clomp**/
- **qsub –I –V –lnodes=2:mem1gb,walltime=00:30:00 –d `pwd`**
  (interactive batch environment on 2 nodes for 30 min.)
- **module switch compiler/intel9.0 compiler/intel9.1**
- **. ␣ init_clomp** (setup of licenses and kmp_cluster.ini in current dir.)

Compilation

- Compiling the application with Intel® Cluster OpenMP:
  icc –cluster–openmp –o *my_prog my_prog***.c**
  icpc –cluster–openmp –o *my_prog my_prog***.cpp**
  or
  ifort –cluster–openmp –o *my_prog my_prog***.f** or **.f90**

Execution

- **export OMP_NUM_THREADS=4**
- **./***my_prog*

Intel® Cluster OpenMP – Practical (on cacau.www.de)

- **init_clomp** must be started in a batch environment:

```
# synopsis:
#   .  init_clomp
export INTEL_LICENSE_FILE=/cacau/HLRS/hlrs/hpcintel/clomp_license.lic
echo '--processes=`cat $PBS_NODEFILE | wc -l` \
      '--process_threads=2' \
      '--hostlist=`cat $PBS_NODEFILE | sed -e 's/$/,/'` \
     | sed -e 's/, /,/g' -e 's/,$//'  >  kmp_cluster.ini
\ln -f kmp_cluster.ini .kmp_cluster
export KMP_CLUSTER_PATH=`pwd`
ulimit -s unlimited
```

- It produces **kmp_cluster.ini** in the local working directory, e.g.,

```
--processes=3  --process_threads=2  --hostlist=noco023.nec,noco024.nec,noco025.nec
```

- and **.kmp_cluster** for access from everywhere via **$KMP_CLUSTER_PATH**

Intel® Cluster OpenMP – Practical (on cacau.www.de)

- **Examples**
  - **cd exa1**
    - No modifications
    - **icc –cluster–openmp –o exa1 c_ex1.c**
    - **ifort –cluster–openmp –o exa1 f_ex1.f**
    - **./exa1**
    - Checking the number of threads → should be 4 (on 2 nodes with each 2 threads)
  - **cd ../exa2**
    - Declaration of sharable variables is necessary
    - **icc –cluster–openmp –clomp–sharable–propagation –ipo –o exa2 c_ex2.c**
    - **ifort –cluster–openmp –clomp–sharable–propagation –ipo –o exa2 f_ex2.f**
    - → There are variables used as shared, but not declared as sharable:
    - C: **#pragma intel omp sharable(xdim, ydim, n)** in main
    - **r_tmp2 = … kmp_sharable_malloc(….);**
    - Fortran: **!dir$ omp sharable(xdim, ydim)** in module dims
    - **!dir$ omp sharable(n)** in the main program
    - **icc –cluster–openmp –o exa2 c_ex2.c**
    - **ifort –cluster–openmp –o exa2 f_ex2.f**
    - **./exa2**

Intel® Cluster OpenMP – Practical (on cacau.www.de)

- **cd ../exa3** → Declaration of sharable variables is necessary

C++:
- **export KMP_DISJOINT_HEAPSIZE=2M** (this test does not really help ☹)
- **icpc –cluster–openmp –g –o exa3 cpp_ex3.cpp**
- **./exa3** → seg fault → ip=…. → **addr2line –e exa3 ….** → cpp_ex3.cpp:<line number>
- Add: **#include <kmp_sharable.h>**
- Substitute twice: **vector<House*>**
  → **vector<House*,kmp_sharable_allocator<House*> >**
- Substitute everywhere: **new** → **new kmp_sharable**
- **icpc –cluster–openmp –o exa3 cpp_ex3.cpp**

Fortran:
- **ifort –cluster–openmp –clomp–sharable–propagation –ipo –o exa3 f_ex3.f**
- → There are variables used as shared, but not declared as sharable:
- After the declaration, add: **!dir$ omp sharable(niter)**
- Add with correct common block name: **!dir$ omp sharable(/matrices/)**
- Substitute call run(niter-1) by **ntemp = niter–1**
- **call run(ntemp)**
- Declare **integer ntemp** and add: **!dir$ omp sharable(ntemp)**
- **ifort –cluster–openmp –o exa3 f_ex3.f**

./exa3 → Now it should working

Intel® Cluster OpenMP – Practical (on cacau.www.de)

- **Examples**
  - **cd heat**
    - No modifications
    - **icc –cluster–openmp –Dimax=3000 –Dkmax=3000 –Ditmax=10 –o heat heats2_x.c**
    - **ifort –cluster–openmp –Dimax=3000 –Dkmax=3000 –Ditmax=10 –o heat heats2_x.f**
    - **./heat**

Appendix

- Intel® Compilers with Cluster OpenMP – Consistency Protocol – Examples

Intel® Compilers with Cluster OpenMP – Consistency Protocol – Examples

Notation

- ..=A[i] Start/End — Start/end a read on element i on page A
- A[i]=.. Start/End — Start/end a write on element i on page A, trap to library
- Twin(A) — Create a twin copy of page A
- WriteNotice(A) — Send write notice for page A to other processors
- DiffReq_A_n(s:f) — Request diffs for page A from node n between s and f
- Diff_A_n(s:f) — Generate a diff for page A in writer n between s and where s and f are barrier times. This also frees the twin for page A.

Courtesy of J. Cownie, Intel

Exa. 1

Node 0	Node 1
Barrier 0	**Barrier 0**
A[1]=.. Start	
Twin(A)	
A[2]=.. End	
	A[5]=.. Start
	Twin(A)
	A[5]=.. End
Barrier 1	**Barrier 1**
WriteNotice(A)	Writenotice(A)
A[5]=.. Start	
Diffreq_A_1(0:1)->	
	<-Diff_A_1(0:1)
Apply diffs	
A[5]=.. End	
Barrier 2	**Barrier 2**
WriteNotice(A)	

Courtesy of J. Cownie, Intel

Exa. 2

Node 0	Node 1	Node 2
Barrier 0	**Barrier 0**	**Barrier 0**
A[1]=.. Start		
Twin(A)		
A[1]=.. End		
Barrier 1	**Barrier 1**	**Barrier 1**
WriteNotice(A)		
A[2]=.. (no trap to library)		
Barrier 2	**Barrier 2**	**Barrier 2**
(No WriteNotice(A) required)		
A[3]=.. (no trap to lib)		
	..=A[1] Start	
	<-Diffreq_A_0(0:2)	
Diff_A_0(0:2)->		
	Apply diffs	
	..=A[1] End	
Barrier 3	**Barrier 3**	**Barrier 3**
(no WriteNotice(A) required because diffs were sent after the A[3]=..)		
A[1]=.. Start		
Twin(A)		
Barrier 4	**Barrier 4**	**Barrier 4**
WriteNotice(A)		
		..=A[1] Start
		<- Diffreq_A_0(0:4)
Create Diff_A_0(2:4) send Diff_A_O(0:4)->		
		Apply diffs
		..=A[1] End

Courtesy of J. Cownie, Intel

Exa. 3 (start)

Node 0	Node 1	Node 2	Node 3
Barrier 0	**Barrier 0**	**Barrier 0**	**Barrier 0**
A[1]=.. Start	A[5]=.. Start		
Twin(A)	Twin(A)		
A[1]=.. End	A[5]=.. End		
Barrier 1	Barrier 1	Barrier 1	Barrier 1
WriteNotice(A)	WriteNotice(A)		
A[2]=.. Start	A[1]=.. Start		
Diffreq_A_1(0:1)->	<-Diffreq_A_0(0:1)		
Diff_A_0(0:1)->	<-Diff_A_1_(0:1)		
Apply diff	Apply diff		
Twin(A)	Twin(A)		
A[2]=.. End	A[1]=.. End		
Barrier 2	**Barrier 2**	**Barrier 2**	**Barrier 2**
WriteNotice(A)	WriteNotice(A)		
A[3]..= Start	A[6]..= Start		
Diffreq_A_1(1:2)->	<-Diffreq_A_A(1:2)		
Diffs_A_0(1:2)->	<-Diffs_A_1(1:2)		
Apply diffs	Apply diffs		
Twin(A)	Twin(A)		
A[3]=.. End	A[6]=.. End		
		..=A[1] Start	
		<-Diffreq_A_0(0:2)	
		<-Diffreq_A_1(0:2)	
Create Diff_A_0(1:2)	Create Diff_A_1(1:2)		
Send Diff_A_0(0:2)->	Send Diff_A_1(0:2)->		
		Apply all diffs	
		..=A[1] End	

Courtesy of J. Cownie, Intel

Node 0	Node 1	Node 2	Node 3
Barrier 3	**Barrier 3**	**Barrier 3**	**Barrier 3**
Writenotice(A)	Writenotice(A)		
A[1]=.. Start			
Diffreq_A_1(2:3)->			
	<-Diffs_A_1_(2:3)		
Apply diffs			
Twin(A)			
A[1]..= End			
Barrier 4	**Barrier 4**	**Barrier 4**	**Barrier 4**
Writenotice(A)			
			..=A[1] Start
			<-Diffreq_A_0(0:4)
			<-Diffreq_A_1(0:4)
Create Diff_A_0(3:4)	Create Diff_A_1(2:4)		
Send Diff_A_0(0:4)->	Send Diff_A_1(0:4)->		
			Apply diffs
			..=A[1] End

These examples may give an impression of the overhead induced by the Cluster OpenMP consistency protocol.

Courtesy of J. Cownie, Intel