Intel Cluster OpenMP™

Georg Hager
Regionales Rechenzentrum Erlangen
Universität Erlangen-Nürnberg

HLRS Parallel Tools Workshop
09./10.07.2007
Outline

- What is it?
- Process Model & Sharability
- Consistency Protocol
- Compiler Options
- Running a Cluster OpenMP Program

Benchmarks
- Low-level
- Heat Conduction
- Lattice-Boltzmann Code

Introductory material provided by Jim Cownie and Larry Meadows of Intel
What is it?

- Cluster OpenMP supports running **OpenMP code on the nodes of a cluster**
  - Compiler extensions and runtime library
- Part of Intel compilers since version 9.1
  - Separate license required
- “Shared memory” structure is simulated in software

**Pros**

- All the good things from OpenMP + ccNUMA… 😊
- Memory bandwidth, automatic page replication/migration
- Inexpensive hardware a.o.t. OpenMP on shared memory

**Cons**

- Synchronization very expensive
- Remote access very expensive
- Sharability sometimes not recognized by compiler
- False sharing on page basis is **catastrophic** for performance
Cluster OpenMP Process Model

- Usually one process per node, one OpenMP thread per core
- Cluster OpenMP global thread ID runs from 0 to (number of nodes)x(threads per node)
- “Hybrid” programming in disguise
Cluster OpenMP Memory Model

- If multiple threads access a variable, it must be **sharable**
- A sharable variable must be allocated in the **sharable region of address space**
Cluster OpenMP Sharability

- OpenMP shared data must be made sharable in Cluster OpenMP
  - implicitly (by compiler)
  - explicitly (by options or directives)

  ```
  #pragma intel omp sharable(a)  // C/C++
  !dir$ omp sharable(a)  ! Fortran
  ```

- No single system image
  - No shared file pointers
  - No shared sockets
  - … or any other shared OS resources
Made Sharable by the Compiler

- Routine-local stack variables within the scope of the routine

```c
void func(void) {
    int a;
    ...
#pragma omp parallel shared(a)
}
```

- Call by value parameters inside the scope of the called routine

```c
void func(int a) {
    ...
#pragma omp parallel shared(a)
}
```
Making Fortran Data Sharable

- **Fortran COMMON blocks:**
  - by directive: `!dir$ omp sharable (/BLK/)`
  - by compiler option: `-clomp-sharable-commons`
  - `common /BLK/ a(100)`

- **Fortran local SAVE data:**
  - by directive: `!dir$ omp sharable(a)`
  - by compiler option: `-clomp-sharable-localsaves`
  - `real a(100)`
  - `save a`

- **Fortran Module variables:**
  - by directive: `!dir$ omp sharable(a)`
  - by compiler option: `-clomp-sharable-modvars`
  - `module m`
  - `real a(100)`
Making Dynamic Data Sharable

- **Fortran: by directive**

  ```fortran
  use omp_lib
  real, allocatable :: a(:)
  !dir$ omp sharable(a)

  allocate a(size)
  ```

- **C: substitutes for `malloc()`/`free()`**

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

All objects of a class A become sharable by inheriting from class kmp_sharable_base:

```cpp
class A : public kmp_sharable_base {...};
```

Make a single dynamic object sharable:

```cpp
A *ptr = new (kmp_sharable) A;
```

Sharable STL containers:

```cpp
vector<int,kmp_sharable_allocator<int> > *ptr =
new (kmp_sharable) vector<int,
    kmp_sharable_allocator<int> > ;
```
Detecting Missing Sharable Specifications

- For static and stack data
  
  `-cluster-openmp -clomp-sharable-propagation -ipo`

  detects variables which should be sharable at link time:

  `fortcom: Warning: Sharable directive should be inserted by user as `!dir$ omp sharable(n)` in file ... line ...`

- For dynamic data

  use `KMP_DISJOINT_HEAPSIZE=<size>` shell variable to provoke segfaults on non-sharable heap data which is used in a shared way
A Simple Cluster OpenMP Program

```
#include <omp.h>
static int a;
#pragma intel omp sharable(a)

int main()
{
    a = 0;
#pragma omp parallel
    {
        #pragma omp critical
        a++;
    }
    printf("%d should equal %d\n", 
        omp_get_max_threads(), a);
}
```
Compiling and Running

- **Compile:** `icc -cluster-openmp test.c`

- **Write** `kmp_cluster.ini` **file with details on how to run the binary:**
  
  ```
  --hostfile=./hosts --process_threads=4
  --launch=ssh --sharable-heap=200000000
  --transport=dapl --adapter=OpenIB-cma
  ```

- `./a.out`
Latency Galore: Vector Triad (Woodcrest 3GHz)

\[ A(\cdot) = B(\cdot) + C(\cdot) \times D(\cdot) \]
Heat Conduction Example

- Simple 2D finite difference solver for heat conduction equation

```c
!$OMP parallel do private(dphi,i) reduction(max:dphimax)
schedule(static)
  do k=1,kmax-1
    do i=1,imax-1
      dphi=(phi(i+1,k,t0)+phi(i-1,k,t0)- &
         2.0_8*phi(i,k,t0))*dy2i &
       +(phi(i,k+1,t0)+phi(i,k-1,t0)- &
         2.0_8*phi(i,k,t0))*dx2i
      dphi=dphi*dt
      dphimax=max(dphimax,abs(dphi))
      phi(i,k,t1)=phi(i,k,t0)+dphi
    enddo
  enddo
!$OMP end parallel do
```
Communication vs Computation
4000x4000 Grid

Efficiency vs Cores

- **AMD Opteron**
- **CLOMP IBand**
- **CLOMP TCPoE**
- **Opteron noNUMA**

Cluster OpenMP
Communication vs. Computation
Favourable Case
Communication vs. Computation
Unfavourable Case

![Graph showing communication vs. computation efficiency](image)

- AMD Opteron
- CLOMP IBand
- CLOMP TCPoE
Communication vs. Computation
Good Problem Sizes

CLOMP IBand 500 x NX, 8 threads

Efficiency vs. NX

NX: 4000, 8000, 16000, 32000, 64000, 128000, 256000
Efficiency: 0, 0.2, 0.4, 0.6, 0.8, 1, 1.2
Application: Lattice Boltzmann Method

Numerical Method for Simulation of Fluids

Stream-Collide (Pull-Method)
Get the distributions from the neighboring cells in the source array and store the relaxed values to one cell in the destination array.

Collide-Stream (Push-Method)
Take the distributions from one cell in the source array and store the relaxed values to the neighboring cells in the destination array.

D3Q19 model:

Two Grids:
Compressed Grid (not used here):
double precision
   f(0:xMax+1,0:yMax+1,0:zMax+1,0:18,0:1)
 !$OMP PARALLEL DO
 do z=1,zMax
   do y=1,yMax
     do x=1,xMax
       if( fluidcell(x,y,z) ) then
         LOAD f(x,y,z, 0:18,t)
         ...
         Relaxation (complex computations)...
         SAVE f(x ,y ,z , 0,t+1)
         SAVE f(x+1,y+1,z , 1,t+1)
         SAVE f(x ,y+1,z , 2,t+1)
         SAVE f(x-1,y+1,z , 3,t+1)
         ...
         SAVE f(x ,y-1,z-1,18,t+1)
       endif
     enddo
   enddo
 enddo
 enddo
LBMKernel

- Scalability beyond 2 nodes was very bad with standard code
- proper choice of geometry (long thin channel) can restore scalability
  - not a general solution
- **Solution**: bounceback (boundary) routine was not properly optimized for local access
  - on ccNUMA, this is a negligible effect for small obstacle density ($n^2$)
  - on CLOMP, it is devastating
- **Still**: indexing has significant impact on performance
  - "push" vs. "pull" algorithm
  - parallelized dimension should be the outermost one to minimize false sharing: $(i,j,v,t,k)$ better than $(l,j,k,v,t)$
- Might profit from ghost layers, but is this still OpenMP???
Influence of Bounceback and Push vs. Pull for 128x64x128 and (i,j,k,v,t) layout

MLUPS

- push
- pull
- pull nobb
- orig (push wrong BB)
- push nobb optidx

- 1T
- 2T
- 4T

(l,j,v,t,k) layout
Cluster OpenMP
Wrap-Up

- Interesting model, fun to play with
- Crucial to get the right IB setup
- Benefits
  - Improved memory bandwidth
  - Inexpensive hardware
- Drawbacks
  - Huge latency penalties
  - Communication vs. Computation
  - Large memory footprint

- “Minimalistic” MPI?
Hand-On

- qsub -l nodes=2:dgrid,walltime=01:00:00 -I -X
- module load compiler/intel-10.0
- cd CLOMP/exa1
- cp $PBS_NODEFILE hosts
- echo "--process_threads=1 --launch=ssh \ 
--hostfile=hosts" > kmp_cluster.ini

Then either:

- ifort -cluster-openmp f_ex1.f
- icc -cluster-openmp c_ex1.c

./a.out
a= 2. It should be 2.