NUMA Control for Hybrid Applications

Hang Liu
TACC
September 13, 2011
Hybrid Applications

• Typical definition of hybrid application
  – Uses both message passing (MPI) and a form of shared memory algorithm (OMP)
  – Runs on multicore systems
  – Multicore systems have multilayered, complex memory architecture

• Hybrid programming does not guarantee optimal performance
  – But it is required for very large core counts (MPI limitation)
  – Actual performance is heavily application dependent

• Non-Uniform Memory Access
  – Multiple memory levels
  – Different access latencies for different levels
  – Complicated by asymmetries in multisocket, multicore systems
  – More responsibility on the programmer to make application efficient
Multisocket, Multicore Systems

- A limited number of processors $N$ have access to a common pool of shared memory
- Access to memory from each core is not uniform (NUMA)
- Hybrid applications have two levels of parallelism
  - Intra-Node (multithread, OMP)
  - Inter-Nodes (message passing, MPI)
- Efficient use of NUMA control is key to performance in hybrid applications
Modes of Hybrid Operation

16 MPI Tasks
4 MPI Tasks
4 Threads/Task
1 MPI Tasks
16 Threads/Task

Pure MPI
1 MPI Task
Thread on each Core

Master Thread of MPI Task
- MPI Task on Core
- Master Thread of MPI Task
- Slave Thread of MPI Task
Needs for NUMA Control

• Asymmetric multi-core configuration on node requires better control on core affinity and memory policy.
  – Load balancing issues on node

• Slowest CPU/core on node may limit overall performance
  – use only balanced nodes, or
  – employ special in-code load balancing measures

• Applications performance can be enhanced by specific arrangement of
  – tasks (process affinity)
  – memory allocation (memory policy)
NUMA Operations

- Each thread is executed by a core and has access to a certain memory space
  - Core assigned by process affinity
  - Memory allocation assigned following memory policy

- Users can manually control the affinity and memory policy using NUMA operations
  - NUMA Control is managed by the kernel (default).
  - Default NUMA Control settings can be overridden with `numactl`.
NUMA Operations

• Ways Process Affinity and Memory Policy can be changed:
  – Dynamically on a running process (knowing process id)
  – At process execution (with wrapper command)
  – Within program through F90/C API

• Users can alter Kernel Policies by manually setting Process Affinity and Memory Policy
  – Users can assign their own processes onto specific cores.
  – Avoid overlapping of multiple processes
numactl Syntax

- Affinity and Policy can be changed externally through `numactl` at the socket and core level.

```plaintext
Command:  numactl <options> ./a.out
```

**Socket References**

2:  3:  1:  0:

- 0,1,2,3
- 4,5,6,7

**Core References**

8,9,10,11:  12,13,14,15

- 0,1,2,3
- 4,5,6,7
# numactl Options

<table>
<thead>
<tr>
<th>Category</th>
<th>Command</th>
<th>Option</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket Affinity</td>
<td>numactl</td>
<td>-N</td>
<td>{0,1,2,3}</td>
<td>Only execute process on cores of this (these) socket(s).</td>
</tr>
<tr>
<td>Memory Policy</td>
<td>numactl</td>
<td>-l</td>
<td>{no argument}</td>
<td>Allocate on current socket.</td>
</tr>
<tr>
<td>Memory Policy</td>
<td>numactl</td>
<td>-i</td>
<td>{0,1,2,3}</td>
<td>Allocate round robin (interleave) on these sockets.</td>
</tr>
<tr>
<td>Memory Policy</td>
<td>numactl</td>
<td>--preferred=</td>
<td>{0,1,2,3}</td>
<td>Select only one Allocate on this socket; fallback to any other if full.</td>
</tr>
<tr>
<td>Memory Policy</td>
<td>numactl</td>
<td>-m</td>
<td>{0,1,2,3}</td>
<td>Only allocate on this (these) socket(s).</td>
</tr>
<tr>
<td>Core Affinity</td>
<td>numactl</td>
<td>-C</td>
<td>{0,1,2,3, 4,5,6,7, 8,9,10,11, 12,13,14,15}</td>
<td>Only execute process on this (these) Core(s).</td>
</tr>
</tbody>
</table>
Memory Policies

- MPI – local is best
- SMP – Interleave best for large, completely shared arrays
- SMP – local is best for private arrays
- Once allocated, a memory structure’s is fixed

Memory: Socket References
Hybrid Runs with NUMA Control

• A single MPI task (process) is launched and becomes the “master thread”.
• It uses any `numactl` options specified on the launch command.
• When a parallel region forks the slave threads, the slaves inherit the affinity and memory policy of the master thread (launch process).
Hybrid Batch Script 16 threads

- Make sure 1 MPI task is created on each node
- Set number of OMP threads for each node
- Can control only memory allocation
- No simple/standard way to control thread-core affinity

---

**job script** (Bourne shell)

```
... 
#! -pe 1way 192
...
export OMP_NUM_THREADS=16
ibrun numactl –i all ./a.out
```

**job script** (C shell)

```
... 
#! -pe 1way 192
...
setenv OMP_NUM_THREADS 16
ibrun numactl –i all ./a.out
```
**Hybrid Batch Script**  
4 tasks, 4 threads/task

<table>
<thead>
<tr>
<th>job script (Bourne shell)</th>
<th>job script (C shell)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>#! -pe 4way 192</td>
<td>#! -pe 4way 32</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>export OMP_NUM_THREADS=4</td>
<td>setenv OMP_NUM_THREADS 4</td>
</tr>
<tr>
<td>ibrun numa.sh</td>
<td>ibrun numa.csh</td>
</tr>
</tbody>
</table>

**numa.sh**

```bash
#!/bin/bash
export MV2_USE_AFFINITY=0
export MV2_ENABLE_AFFINITY=0
export VIADEV_USE_AFFINITY=0

#TasksPerNode
TPN=`echo $PE | sed 's/way//'`
[ ! $TPN ] && echo TPN NOT defined!
[ ! $TPN ] && exit 1

socket=$(($PMI_RANK % $TPN ))
numactl -N $socket -m $socket ./a.out
```

**numa.csh**

```bash
#!/bin/tcsh
setenv MV2_USE_AFFINITY 0
setenv MV2_ENABLE_AFFINITY 0
setenv VIADEV_USE_AFFINITY 0

#TasksPerNode
set TPN = `echo $PE | sed 's/way//'`
if(! %{TPN}) echo TPN NOT defined!
if(! %{TPN}) exit 0

@ socket = $PMI_RANK % $TPN
numactl -N $socket -m $socket ./a.out
```

**for mvapich2**
Hybrid Batch Script with tacc_affinity

- Simple setup for ensuring evenly distributed core setup for your hybrid runs.
- tacc_affinity is not the single magic solution for every application out there - you can modify the script and replace tacc_affinity with yours for your code.

<table>
<thead>
<tr>
<th>job script (Bourne shell)</th>
<th>job script (C shell)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>#! -pe 4way 192</td>
<td>#! -pe 4way 192</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>export OMP_NUM_THREADS=4</td>
<td>setenv OMP_NUM_THREADS 4</td>
</tr>
<tr>
<td>ibrun tacc_affinity ./a.out</td>
<td>ibrun tacc_affinity ./a.out</td>
</tr>
</tbody>
</table>

• Simple setup for ensuring evenly distributed core setup for your hybrid runs.
• tacc_affinity is not the single magic solution for every application out there - you can modify the script and replace tacc_affinity with yours for your code.
#!/bin/bash

MODE=`/share/sge/default/pe_scripts/getmode.sh`
# First determine "wayness" of PE
myway=`echo $PE | sed s/way//`

# Determine local compute node rank number
if [ x"$MODE" == "xmvapich2_ssh" ]; then
    export MV2_USE_AFFINITY=0
    export MV2_ENABLE_AFFINITY=0
    my_rank=$PMI_ID
elif [ x"$MODE" == "xmvapich1_ssh" ]; then
    export VIADEV_USE_AFFINITY=0
    export VIADEV_ENABLE_AFFINITY=0
    my_rank=$MPIRUN_RANK
else
    echo "TACC: Could not determine MPI stack. Exiting!"
    exit 1
fi
local_rank=$(( $my_rank % $myway ))
# Based on "wayness" determine socket layout on local node
# if less than 4-way, offset to skip socket 0
if [ $myway -eq 1 ]; then
    numnode="0,1,2,3"
# if 2-way, set 1st task on 0,1 and second on 2,3
elif [ $myway -eq 2 ]; then
    numnode="(( 2 * $local_rank )),(( 2 * $local_rank + 1 ))"
elif [ $myway -lt 4 ]; then
    numnode=$(( $local_rank + 1 ))
# if 4-way to 12-way, spread processes equally on sockets
elif [ $myway -lt 13 ]; then
    numnode=$(( $local_rank / ( $myway / 4 ) ))
# if 16-way, spread processes equally on sockets
elif [ $myway -eq 16 ]; then
    numnode=$(( $local_rank / ( $myway / 4 ) ))
# Offset to not use 4 processes on socket 0
else
    numnode=$(( ($local_rank + 1) / 4 ))
fi
#echo "TACC: Running $my_rank on socket $numnode"
exec numactl -c $numnode -m $numnode $*
Summary

• NUMA control ensures hybrid jobs to run with optimal core affinity and memory policy.
• Users have global, socket, core-level control for process and threads arrangement.
• Possible to get great return with small investment by avoiding non-optimal core/memory policy.