NUMA Control for Hybrid Applications

Hang Liu
TACC
February 7th, 2011

Hybrid Applications

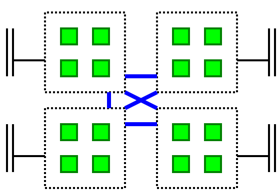
- Typical definition of hybrid application
  - Uses both message passing (MPI) and a form of shared memory algorithm (OMP)
  - Runs on multicore systems
  - Multicore systems have multilayered, complex memory architecture
- Hybrid programming does not guarantee optimal performance
  - But it is required for very large core counts (MPI limitation)
  - Actual performance is heavily application dependent
- **N**on-**U**niform **M**emory **A**ccess
  - 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

Modes of Hybrid Operation

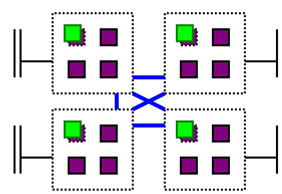
Pure MPI

1 MPI Task
Thread on each Core

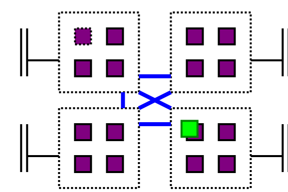
16 MPI Tasks



4 MPI Tasks
4Threads/Task



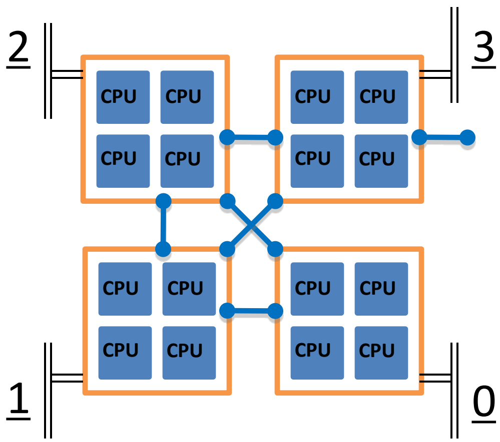
1 MPI Tasks
16 Threads/Task



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 by memory policy

- The control of process affinity and memory policy using NUMA operations
  - NUMA Control is managed by the kernel (default).
  - Default NUMA Control settings can be overridden with **numaclt**.

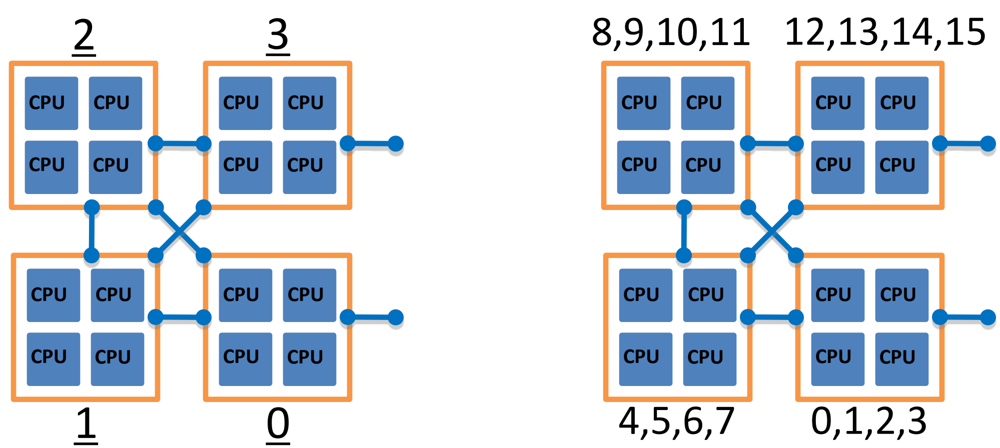
NUMA Operations

- Ways Process Affinity and Memory Policy can be managed:
  - 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.

Command: numactl <options> ./a.out

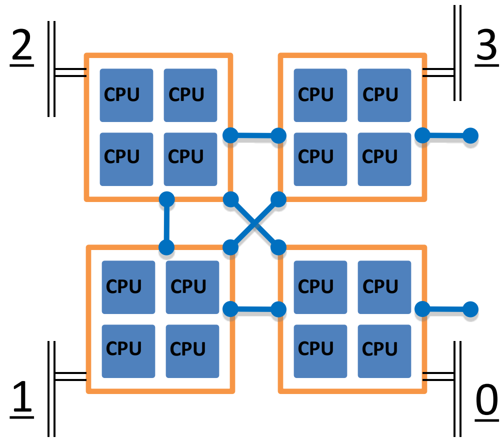


Socket References   Core References

numactl Options on Ranger

	cmd	option	arguments	description
Socket Affinity	numactl	**-N**	{0,1,2,3}	Only execute process on cores of this (these) socket(s).
Memory Policy	numactl	**-l**	{no argument}	Allocate on current socket.
Memory Policy	numactl	**-i**	{0,1,2,3}	Allocate round robin (interleave) on these sockets.
Memory Policy	numactl	**--preferred=**	{0,1,2,3} select only one	Allocate on this socket; fallback to any other if full .
Memory Policy	numactl	**-m**	{0,1,2,3}	Only allocate on this (these) socket(s).
Core Affinity	numactl	**-C**	{0,1,2,3, 4,5,6,7, 8,9,10,11, 12,13,14,15}	Only execute process on this (these) Core(s).

Memory Policies



Memory: Socket References

- 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

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)	**job script** (C shell)
...	...
#! -pe 1way 192	#! -pe 1way 192
...	...
export OMP_NUM_THREADS=16	setenv OMP_NUM_THREADS 16
ibrun numactl –i all ./a.out	ibrun numactl –i all ./a.out

Hybrid Batch Script 4 tasks, 4 threads/task

for mvapich2

job script (Bourne shell)	**job script** (C shell)
... #! -pe 4way 192 ... export OMP_NUM_THREADS=4 ibrun numa.sh	... #! -pe 4way 32 ... setenv OMP_NUM_THREADS 4 ibrun numa.csh

numa.sh

```
#!/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

```
#!/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
```

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.

job script **(Bourne shell)**	**job script** **(C shell)**
...	...
#! -pe 4way 192	#! -pe 4way 192
...	...
export OMP_NUM_THREADS=4	setenv OMP_NUM_THREADS 4
ibrun tacc_affinity ./a.out	ibrun tacc_affinity ./a.out

tacc_affinity

```
#!/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
```

tacc_affinity (cont'd)

```
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.