Lab: Hybrid Programming and NUMA Control

Steve Lantz

Introduction to Parallel Computing
May 24, 2011

Based on materials developed by by Kent Milfeld at TACC

Cornell University
Center for Advanced Computing

What You Will Learn

- How to use numactl in the execution of serial, threaded, and 4xN hybrid (i.e., 4 MPI tasks, each with N threads) codes
- How to structure communications in a 2x16 hybrid code that involves threaded MPI calls between 2 nodes
  - MPI calls from serial region
  - MPI calls from master thread in a parallel region
  - MPI calls from all threads in a parallel region
- How to measure the performance of the above codes
- The performance implications of using numactl and threaded MPI
  - Location of data is important in serial codes
  - Initialization of data is important in threaded codes
  - For less than 16-way, MPI executables need to be assigned to sockets

Getting Started

- Untar the file numahybrid.tar
  - cd ~ (start in your home directory)
  - tar xvf ~train100/labs/numahybrid.tar (extract files)
  - cd numahybrid

Cornell University
Center for Advanced Computing

numactl_serial

- Run the memory intensive daxpy program on four different sockets using local, interleave and off-socket-memory policies.
  - Use the commands below to make the daxpy executable and run it with numa control commands.
  - See the job script and the table on the next page for the numa options.
  - Run the job and report the times and relative performance.
- Procedure:
  - cd numactl_serial (change directory to numactl_serial)
  - module unload mvapich; module swap pgi intel; module load mvapich
  - make
  - qsub job (submits job)

numactl_serial – Results

- From the job output fill in the table.

Command	Time (secs)
numactl -l -C 0	
numactl -l -C 1	
numactl -l -C 2	
numactl -l -C 3	
numactl -i all -C 0	
numactl -i all -C 1	
numactl -i all -C 2	
numactl -i all -C 3	
numactl -m 3 -C 0	

Rank the performance of local, interleave, and off-socket-memory policies, from best to poorest

1.)

2.)

3.)

Cornell University
Center for Advanced Computing

numactl_alloc

- The daxpy algorithm is parallelized to run as 4 threads. It is run on 4 different sockets through the code statements:

```
ithread = OMP_GET_THREAD_NUM()*4
call f90_setaffinity(ithread)
```

  - In master_alloc_daxpy, the a, b, and c matrices are allocated preferentially on the default socket for the master thread (0).
  - In thread_alloc_daxpy, sections of a, b, and c are allocated where the threads are executing (cores 0,4,8,12 on sockets 0,1,2,3).
- Procedure:
  - cd numactl_alloc (change directory to numactl_alloc)
  - if you have done this already, don't do it again:
    module unload mvapich; module swap pgi intel; module load mvapich
  - make (note, must link with -lnuma to obtain set_mempolicy)
  - qsub job (submits job)

numactl_alloc – Results

- From the job output fill in the table.

Command	Time (secs)
master_alloc_daxpy	
thread_alloc_daxpy	

Rank the performance of master-allocated memory vs. thread-allocated, from best to poorest

1.)

2.)

Cornell University
Center for Advanced Computing

numactl_4x1, numactl_4x4

- Run the daxpy program as 4 tasks in a node (4x1) and 4 tasks with 4 threads in a node (4x4), following the instructions below.
  - Use the commands below to make the daxpy executable and run it with numa control commands.
  - See the job script and the table on the next page for the numa options.
  - Run the job and report the times and relative performance.
- Procedure:
  - cd numactl_4x1 or numactl_4x4 (change directory as needed)
  - if you have done this already, don't do it again:
    module unload mvapich; module swap pgi intel; module load mvapich
  - make
  - qsub job (submits job)

numactl_4x1, numactl_4x4 – Results

- From the job output fill in the tables.

Command (4x1)	Time (secs)
<no numactl>	
numactl -l	
numactl -i all	
numactl tacc_affinity	

Rank 4x1 performance from best to poorest

1.)
2.)
3.)
4.)

Command (4x4)	Time (secs)
<no numactl>	
numactl -l	
numactl -i all	
numactl tacc_affinity	

Rank 4x4 performance

1.)
2.)
3.)
4.)

Cornell University
Center for Advanced Computing

What's the Explanation?

- This is a bandwidth-limited code, so the best results are achieved when executions are distributed across all sockets
- For both 4x1 and 4x4 cases, the default kernel policy puts all tasks on socket 0; tweaking the memory allocation doesn't help much ☹
- tacc_affinity spreads the tasks across sockets, which is 2-4x faster
  - 4x1 case is 2-3x faster
  - 4x4 case is 4x faster because the default affinity puts ALL threads on a single socket

What is tacc_affinity?

It's a script: /share/sge6.2/default/pe_scripts/tacc_affinity

```
#!/bin/bash
MODE=`/share/sge6.2/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 ))
...
```


Cornell University
Center for Advanced Computing

What is tacc_affinity? – Part 2

```
# 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 /usr/bin/numactl -c $numnode -m $numnode $*
```

Communications in Hybrid Codes

- The tmpi (threaded mpi) code illustrates different ways of doing point-to-point and broadcast communications in a hybrid code. Using both mvapich and openmpi, we will:
  - check to make sure the code performs correctly
  - measure the cost for sending a single large message in the serial region
  - compare the cost for sending 16 small messages in the parallel region
- Procedure:
  - cd threaded_mpi
  - if you have done this already, don't do it again:
    module unload mvapich; module swap pgi intel; module load mvapich
  - ./build.sh (this builds tmpi.mvapich1 and tmpi.openmp)

Cornell University
Center for Advanced Computing

Hybrid Job Script

Script for 10 interactive minutes of 2 nodes (=32/16), 1 task per node (1way), 2 tasks total, in the development queue. 16 threads (OMP_NUM_THREADS 16) are launched on each node.

```
#!/bin/tcsh
#                          # use bash shell
#$ -V                      # inherit submission environment
#$ -cwd                    # use submission directory
#$ -N threadedmpi          # jobname (threadedmpi)
#$ -j y                    # stdout/err combined
#$ -o $JOB_NAME.o$JOB_ID   # output name jobname.ojobid
#$ -pe 1way 32             # 1 task/node, 32 cores total
#$ -q development          # queue name !! use normal
#$ -l h_rt=00:10:00        # request 10 minutes
#$ -A TACCacct             # Accounting: training project
 set echo                  # echo cmds, use "set -x" in sh

 setenv MY_NSLOTS 2
 setenv OMP_NUM_THREADS 16
 ibrun ./tmpi < input
```

If # of tasks is not equal to wayness*total_cores/16, set value here.

Submit the Batch Job

% qsub job

```
...
------- Welcome to TACC's Ranger System, an NSF TeraGrid Resource -------
...
Your job 18073 ("threadedmpi") has been submitted
```

% qstat

```
job-ID  prior   name       user     state submit/start at     queue         slots
-----------------------------------------------------------------------------------
  18075 0.00001 threadedmp milfeld  r     01/17/2008 22:48:54 normal@i104-408 32
```

% showq

```
...
```


Communication from Serial Region

```
include "mpif.h"
…
call MPI_Init_thread(MPI_THREAD_MULTIPLE, iprovided,ierr)
call MPI_Comm_size(MPI_COMM_WORLD,nranks, ierr)
call MPI_Comm_rank(MPI_COMM_WORLD,irank,ierr)

if(irank == 0) then
   call mpi_send(as,n,MPI_REAL8, 1,9,MPI_COMM_WORLD, ierr)
   call mpi_recv(as,n,MPI_REAL8, 1,1,MPI_COMM_WORLD, istatus,ierr)
else if (irank == 1) then
   call mpi_recv(as,n,MPI_REAL8, 0,9,MPI_COMM_WORLD, istatus,ierr)
   call mpi_send(as,n,MPI_REAL8, 0,1,MPI_COMM_WORLD, ierr)
endif


if(irank .eq. 0) read(*,'(i5)') iread1
call MPI_Bcast(iread1,1,MPI_INTEGER, 0,iwcomm, ierr)
```

"Serial Code"

(don't forget error argument in f90 codes)

Broadcast in Parallel Region

```
!$OMP PARALLEL private(i,ithread,nthreads, icp1, icp2, icpd)


ithread = OMP_GET_THREAD_NUM()

if(ithread == 0) then
   if(irank .eq. 0) read(*,'(i5)') iread2
   call MPI_Bcast(iread2,1,MPI_INTEGER, 0,iwcomm, ierr)
end if
```

Parallel Region

(don't forget error argument in f90 codes)

Point-to-point in Parallel Region

```
!$OMP DO ordered
  do i = 1,nthreads
!$OMP ordered
   if(irank == 0) then
      call mpi_send(as,ns,MPI_REAL8, 1,ithread,MPI_COMM_WORLD, ierr)
      call mpi_recv(as,ns,MPI_REAL8, 1,ithread,MPI_COMM_WORLD, istatus,ierr)
   else if (irank == 1) then
      call mpi_recv(as,ns,MPI_REAL8, 0,ithread,MPI_COMM_WORLD, istatus,ierr)
      call mpi_send(as,ns,MPI_REAL8, 0,ithread,MPI_COMM_WORLD,ierr)
    endif
!$OMP end ordered
  end do
if(irank == 0 .and. ithread == 0) then
     call mpi_send(as,n,MPI_REAL8, 1,ithread,MPI_COMM_WORLD, ierr)
     call mpi_recv(ar,n,MPI_REAL8, 1,ithread,MPI_COMM_WORLD, istatus,ierr)
else if (irank == 1 .and. ithread == 0) then
     call mpi_recv(ar,n,MPI_REAL8, 0,ithread,MPI_COMM_WORLD, istatus,ierr)
     call mpi_send(as,n,MPI_REAL8, 0,ithread,MPI_COMM_WORLD, ierr)
endif
!$OMP barrier
!$OMP END PARALLEL
call mpi_finalize(ierr)
```

Parallel Region

Each Thread Sends (block size = ns)

Not needed with mvapich2

Only Thread 0 Sends (block size =n =16 x ns)

End of Parallel
End of MPI

Hybrid Communication Cost (Output from tmpi)

```
Mvapich1
 Serial Region Ping Pong   (words:secs) 400000:     0.00509
 Serial Region Broadcast          (sec)             0.00002
 Parallel Region Broadcast        (sec)             0.00001
 Parallel region messages:
 One Large message   size:secs 400000 tot time:     0.00561
 16  Small messages  size:secs  25000 tot time:     0.00534

individual times:  0.00034  0.00033  0.00034  0.00033  0.00033  0.00033  0.00033
0.00034  0.00033  0.00033  0.00033  0.00033  0.00033  0.00034  0.00033  0.00033

OpenMPI
 Serial Region Ping Pong   (words:secs) 400000:     0.00501
 Serial Region Broadcast          (sec)             0.00005
 Parallel Region Broadcast        (sec)             0.00001
 Parallel region messages:
 One Large message   size:secs 400000 tot time:     0.00550
 16  Small messages  size:secs  25000 tot time:     0.08949

individual times:  0.08383  0.00037  0.00037  0.00038  0.00037  0.00065  0.00035
0.00036  0.00034  0.00035  0.00033  0.00034  0.00037  0.00035  0.00035  0.00036
```


Cornell University
Center for Advanced Computing

Why the Difference in Results?

- Explanation: mvapich has a special queue service which allows multiple short messages (all having the same destination) to be sent as quickly as one long message!