Cornell University
Center for Advanced Computing

Hybrid Programming with OpenMP and MPI

John Zollweg

Introduction to Parallel Computing on Ranger
May 29, 2009

based on material developed by Kent Milfeld, TACC

HW challenges on Ranger?

- Distributed memory - each node has its own - not readily accessible from other nodes
- Multichip nodes - each node has four chips
- Multicore chips - each chip has four cores
- Memory is associated with chips - more accessible from cores on same chip

How do we deal with NUMA?

- NUMA = Non-Uniform Memory Access
- Distributed memory: MPI
- Shared memory: Threads
  - pthreads
  - OpenMP
- Both: Hybrid programming

Why Hybrid?

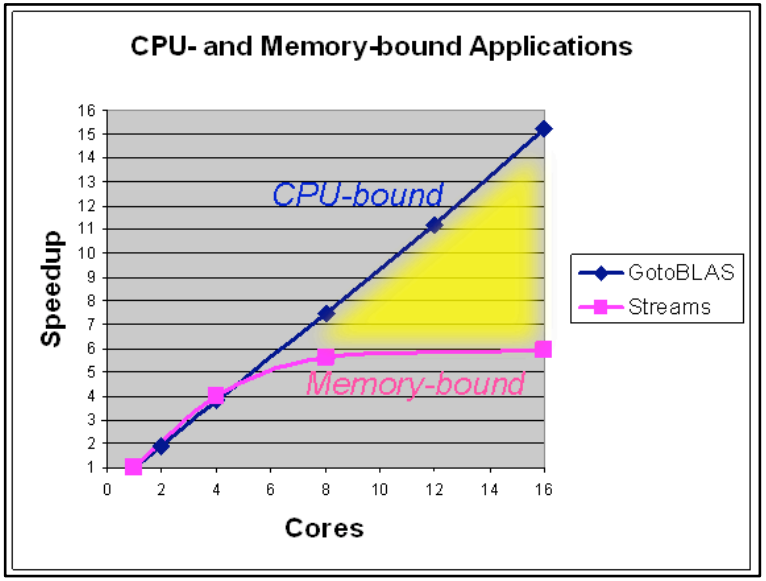
- Eliminates domain decomposition at node
- Automatic memory coherency at node
- Lower (memory) latency and data movement within node
- Can synchronize on memory instead of barrier

Why Not Hybrid?

- Only profitable if on-node aggregation of MPI parallel components is faster as a single SMP algorithm (or a single SMP algorithm on each socket).

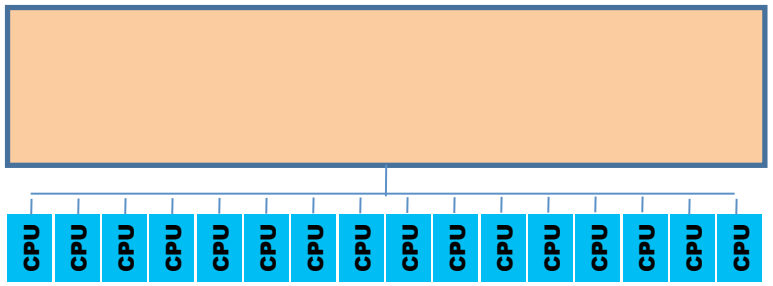
Hybrid - Motivation

- Load Balancing
- Reduce Memory Traffic

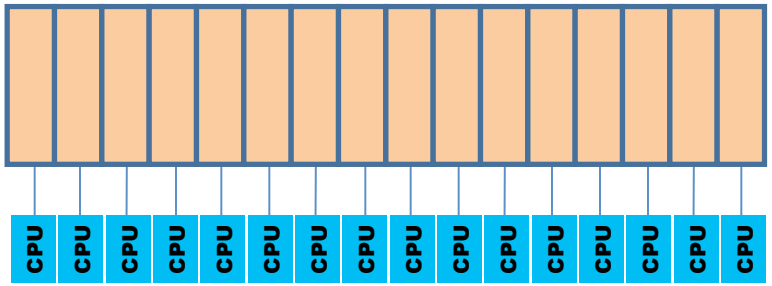


Node Views

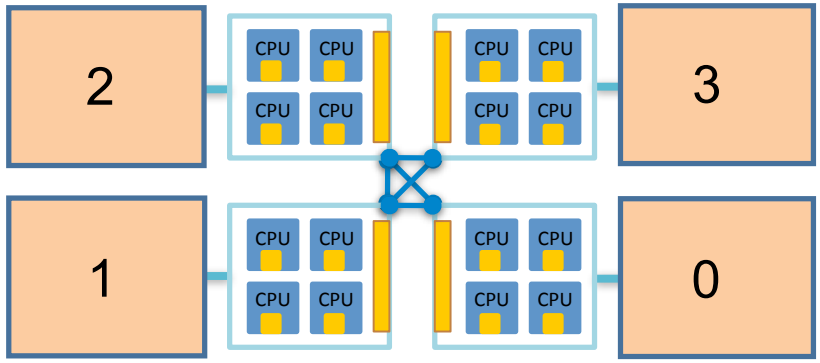
OpenMP



MPI



Process-Affinity Memory-Allocation



NUMA Operations

- Where do threads/processes and memory allocations go?
- If Memory were completely uniform there would be no need to worry about these two concerns. Only for NUMA (non-uniform memory access) is (re)placement of processes and allocated memory (NUMA Control) of importance.
- Default Control: Decided by policy when process exec'd or thread forked, and when memory allocated. Directed from within Kernel.

NUMA CONTROL IS MANAGED BY THE KERNEL.
NUMA CONTROL CAN BE CHANGED WITH NUMACLT.

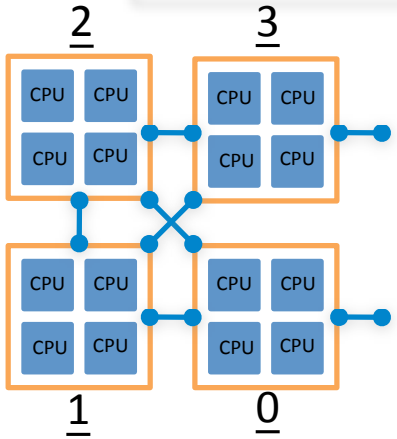
NUMA Operations

- 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 (setting Process Affinity and Memory Policy == PAMPer)
  - Users can PAMPer their own processes.
  - Root can PAMPer any process.
  - Careful, libraries may PAMPer, too!

NUMA Operations

- Process Affinity and Memory Policy can be controlled at socket and core level with numactl.

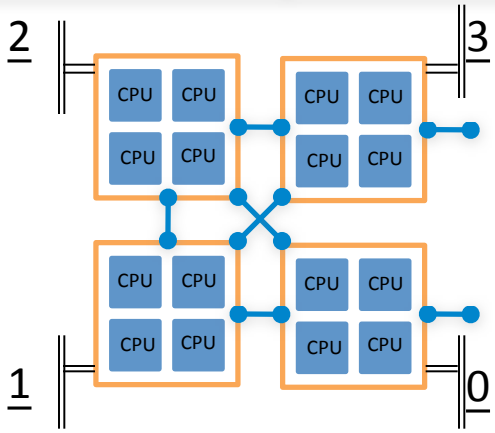
Command: numactl < options socket/core > ./a.out



Process: Socket References

process assignment

-N

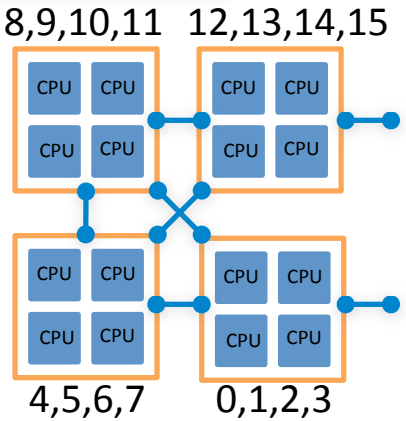


Memory: Socket References

memory allocation

–l –i --preferred –m

(local, interleaved, pref., mandatory)



Process: Core References

core assignment

–C

NUMA Quick Guide

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

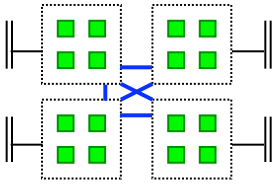
Modes of MPI/Thread Operation

- SMP Nodes
  - Single MPI task launched per node
  - Parallel Threads share all node memory, e.g 16 threads/ node on Ranger.
- SMP Sockets
  - Single MPI task launched on each socket
  - Parallel Thread set shares socket memory, e.g. 4 threads/socket on Ranger
- No Shared Memory (all MPI)
  - Each core on a node is assigned an MPI task.
  - (not really hybrid, but in master/worker paradigm master could use threads)

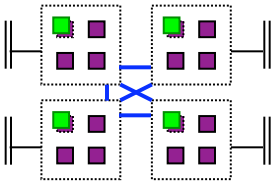
Modes of MPI/Thread Operation

Pure MPI Node — Pure SMP Node

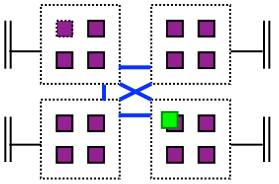
16 MPI Tasks



4 MPI Tasks
4 Threads/Task



1 MPI Task
16 Threads/Task



Master Thread of MPI Task
- MPI Task on Core
- Master Thread of MPI Task
- Worker Thread of MPI Task

SMP Nodes

Hybrid Batch Script 16 threads/node

- Make sure 1 task is created on each node
- Set total number of cores (nodes x 16)
- Set number of threads for each node
- PAMPering at job level
  - Controls behavior for ALL tasks
  - 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

SMP Sockets

Hybrid Batch Script 4 tasks/node, 4 threads/task

- Example script setup for a square (6x6 = 36) processor topology.
- Create a task for each socket (4 tasks per node).
- Set total number of cores allocated by batch (nodes x 16 cores/node).
- Set actual number of cores used with MY_NSLOTS.
- Set number of threads for each task
- PAMPering at task level
  - Create script to extract rank for numactl options, and a.out execution (TACC MPI systems always assign sequential ranks on a node.
  - No simple/standard way to control thread-core affinity

job script (Bourne shell)	**job script** (C shell)
...	...
#! -pe 4way 144	#! -pe 4way 144
...	...
export MY_NSLOTS =36	setenv MY_NSLOTS 36
export OMP_NUM_THREADS=4	setenv OMP_NUM_THREADS 4
ibrun numa.sh	ibrun numa.csh

SMP Sockets

Hybrid Batch Script 4 tasks/node, 4 threads/task

for mvapich2

numa.sh

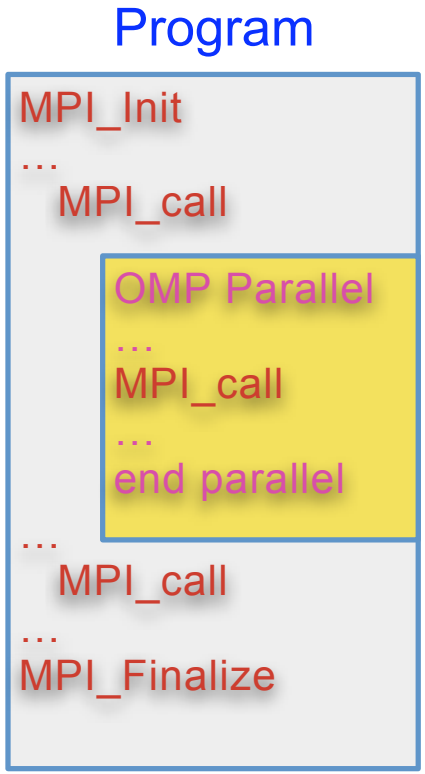
```
#!/bin/bash
export  MV2_USE_AFFINITY=0
export  MV2_ENABLE_AFFINITY=0

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

socket = $(( $PMI_RANK % $TPN ))

exec numactl -N $socket -m $socket ./a.out
```

numa.csh

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

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

@ socket = $PMI_RANK % $TPN

exec numactl -N $socket -m $socket ./a.out
```

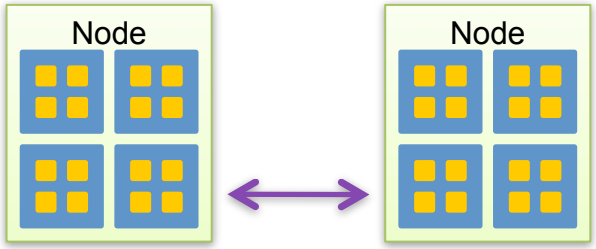
Hybrid – Program Model

- Start with MPI initialization
- Create OMP parallel regions within MPI task (process).
  - Serial regions are the master thread or MPI task.
  - MPI rank is known to all threads
- Call MPI library in serial and parallel regions.
- Finalize MPI



MPI with OpenMP -- Messaging

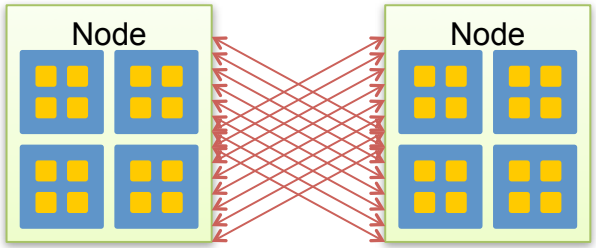
Single-threaded messaging



rank to rank

MPI from serial region or a single thread within parallel region

Multi-threaded messaging



rank-thread ID to any rank-thread ID

MPI from multiple threads within parallel region
Requires thread-safe implementation

Threads calling MPI

- Use MPI_Init_thread to select/determine MPI's thread level of support (in lieu of MPI_Init). MPI_Init_thread is supported in MPI2
- Thread safety is controlled by "provided" types: single, funneled, serialized and multiple
  - Single means there is no multi-threading.
  - Funneled means only the master thread calls MPI
  - Serialized means multiple threads can call MPI, but only 1 call can be in progress at a time (serialized).
  - Multiple means MPI is thread safe.
  - Monotonic values are assigned to Parameters:
    MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE

MPI2 MPI_Init_thread

Syntax:

call MPI_Init_thread(irequired, iprovided, ierr)
int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)
int MPI::Init_thread(int& argc, char**& argv, int required)

Support Levels	Description
MPI_THREAD_SINGLE	Only one thread will execute.
MPI_THREAD_FUNNELED	Process may be multi-threaded, but only main thread will make MPI calls (calls are ”funneled" to main thread). “Default”
MPI_THREAD_SERIALIZE	Process may be multi-threaded, any thread can make MPI calls, but threads cannot execute MPI calls concurrently (MPI calls are ”serialized").
MPI_THREAD_MULTIPLE	Multiple threads may call MPI, no restrictions.

If supported, the call will return provided = required.
Otherwise, the highest level of support will be provided.

Hybrid Coding

Fortran

```
include 'mpif.h'
program hybsimp


call MPI_Init(ierr)
call MPI_Comm_rank (...,irank,ierr)
call MPI_Comm_size (...,isize,ierr)
! Setup shared mem, comp. & Comm

!$OMP parallel do
   do i=1,n
     <work>
   enddo
!  compute & communicate

call MPI_Finalize(ierr)
end
```

C

```
#include <mpi.h>
int main(int argc, char **argv){
 int rank, size, ierr, i;

 ierr= MPI_Init(&argc,&argv[]);
 ierr= MPI_Comm_rank (...,&rank);
 ierr= MPI_Comm_size (...,&size);
//Setup shared mem, compute & Comm

#pragma omp parallel for
   for(i=0; i<n; i++){
     <work>
   }
// compute & communicate

 ierr= MPI_Finalize();
```

MPI Call through Master

- MPI_THREAD_FUNNELED
- Use OMP_BARRIER since there is no implicit barrier in master workshare construct (OMP_MASTER).
- All other threads will be sleeping.

Funneling through Master

Fortran

```
include 'mpif.h'
program hybmas


!$OMP parallel

  !$OMP barrier
  !$OMP master

   call MPI_<Whatever>(…,ierr)
  !$OMP end master

  !$OMP barrier

!$OMP end parallel
end
```

C

```
#include <mpi.h>
int main(int argc, char **argv){
  int rank, size, ierr, i;

#pragma omp parallel
  {
   #pragma omp barrier
   #pragma omp master
   {
    ierr=MPI_<Whatever>(…)
   }

   #pragma omp barrier

  }
}
```

MPI Call within Single

- MPI_THREAD_SERIALIZED
- Only OMP_BARRIER is at beginning, since there is an implicit barrier in SINGLE workshare construct (OMP_SINGLE).
- All other threads will be sleeping.

- (The simplest case is for any thread to execute a single mpi call, e.g. with the “single” omp construct. See next slide.)

Serialize through Single

Fortran

```
include 'mpif.h'
program hybsing

call mpi_init_thread(MPI_THREAD_SERIALIZED,
                     iprovided,ierr)
!$OMP parallel

  !$OMP barrier
  !$OMP single

  call MPI_<whatever>(…,ierr)
  !$OMP end single

  !!OMP barrier

!$OMP end parallel
end
```

C

```
#include <mpi.h>
int main(int argc, char **argv){
int rank, size, ierr, i;
mpi_init_thread(MPI_THREAD_SERIALIZED,
                iprovided)
 #pragma omp parallel
 {
  #pragma omp barrier
  #pragma omp single
  {
   ierr=MPI_<Whatever>(…)
  }

  //pragma omp barrier

 }
}
```

Overlapping Communication and Work

- One core can saturate the PCI-e ←→network bus. Why use all to communicate?
- Communicate with one or several cores.
- Work with others during communication.
- Need at least MPI_THREAD_FUNNELED support.
- Can be difficult to manage and load balance!

Overlapping Communication and Work

Fortran

```
include 'mpi.h'
program hybover


!$OMP parallel

  if (ithread .eq. 0) then
    call MPI_<whatever>(…,ierr)
  else
    <work>
  endif

!$OMP end parallel
end
```

C

```
#include <mpi.h>
int main(int argc, char **argv){
 int rank, size, ierr, i;

 #pragma omp parallel
 {
  if (thread == 0){
    ierr=MPI_<Whatever>(…)
  }
  if(thread != 0){
    work
  }
 }
```

Thread-rank Communication

- Can use thread id and rank in communication
- Example illustrates technique in multi-thread “ping” (send/receive).

Thread-rank Communication

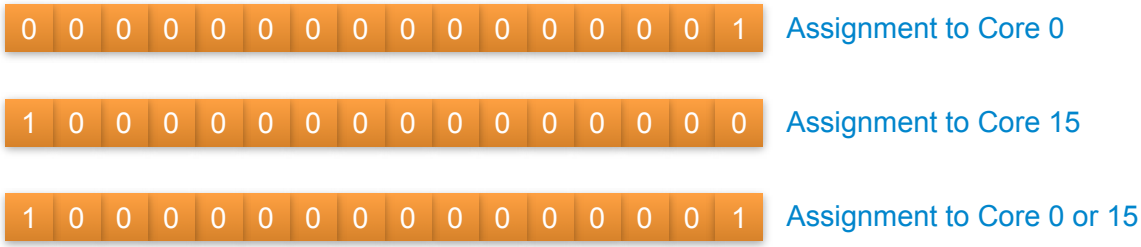
```
…
call mpi_init_thread( MPI_THREAD_MULTIPLE, iprovided,ierr)
call mpi_comm_rank(MPI_COMM_WORLD,  irank, ierr)
call mpi_comm_size( MPI_COMM_WORLD,nranks, ierr)
…
!$OMP parallel private(i, ithread, nthreads)
…
  nthreads=OMP_GET_NUM_THREADS()
  ithread   =OMP_GET_THREAD_NUM()
  call pwork(ithread, irank, nthreads, nranks…)
  if(irank == 0) then
    call mpi_send(ithread,1,MPI_INTEGER, 1,   ithread, MPI_COMM_WORLD, ierr)
  else
    call mpi_recv(          j,1,MPI_INTEGER, 0,   ithread, MPI_COMM_WORLD, istatus,ierr)
    print*, "Yep, this is ",irank," thread ", ithread," I received from ", j
  endif

!$OMP END PARALLEL
end
```

Communicate between ranks.

Threads use tags to differentiate.

NUMA in Code

- Scheduling Affinity and Memory Policy can be changed within code through:
  - sched_get/setaffinity
  - get/set_memorypolicy
- Scheduling: Bits in a mask are set for assignments.



NUMA in Code

- Scheduling Affinity

```
…
#include <spawn.h>
…
int icore=3;
cpu_set_t cpu_mask;
…
CPU_ZERO(      &cpu_mask);
CPU_SET(icore,&cpu_mask);

err = sched_setaffinity(          (pid_t)0  ,
                          sizeof(cpu_mask),
                                 &cpu_mask);
```

C API params/protos

Set core number
Allocate mask

Set mask to zero
Set mask with core #

Set the affinity

Conclusion

- Placement and binding of processes, and allocation location of memory are important performance considerations in pure MPI/OpenMP and Hybrid codes.
- Simple `numactl` commands and APIs allow users to control process and memory assignments.
- 8-core and 16-core socket systems are on the way; even more effort will be focused on process scheduling and memory location.
- Expect to see more multi-threaded libraries.

References

- www.nersc.gov/about/NUG/meeting_info/Jun04/TrainingTalks/NUG2004_yhe_hybrid.ppt
  Hybrid OpenMP and MPI Programming and Tuning (NUG2004),Yun (Helen) He and Chris Ding, Lawrence Berkeley National Laboratory, June 24, 2004.
- www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-2.0/node162.htm#Node162
- services.tacc.utexas.edu/index.php/ranger-user-guide#Process%20Affinity%20and%20Memory%20Policy
- www.mpi-forum.org/docs/mpi2-report.pdf
- www.intel.com/software/products/compilers/docs/fmac/doc_files/source/extfile/optaps_for/common/optaps_openmp_thread_affinity.htm