Hybrid Programming with OpenMP and MPI

Steve Lantz
Senior Research Associate
Cornell CAC

Workshop: Introduction to Parallel Computing on Ranger, July 15, 2010
Based on materials developed by Kent Milfeld at TACC
RAM Arrangement on Ranger

- Many nodes → distributed memory
  - each node has its own local memory
  - not directly addressable from other nodes
- Multiple sockets per node
  - each node has 4 sockets (chips)
- Multiple cores per socket
  - each socket (chip) has 4 cores
- Memory spans all 16 cores → shared memory
  - node’s full local memory is addressable from any core in any socket
- Memory is attached to sockets
  - 4 cores sharing the socket have fastest access to attached memory
Dealing with NUMA

How do we deal with NUMA (Non-Uniform Memory Access)?
Standard models for parallel programs assume a uniform architecture –

• Threads for shared memory
  – parent process uses pthreads or OpenMP to fork multiple threads
  – threads share the same virtual address space
  – also known as SMP = Symmetric MultiProcessing

• Message passing for distributed memory
  – processes use MPI to pass messages (data) between each other
  – each process has its own virtual address space

If we attempt to combine both types of models –

• **Hybrid programming**
  – try to exploit the whole shared/distributed memory hierarchy
Why Hybrid? Or Why Not?

Why hybrid?
• Eliminates domain decomposition at node level
• Automatic memory coherency at node level
• Lower (memory) latency and data movement within node
• Can synchronize on memory instead of barrier

Why not hybrid?
• An SMP algorithm created by aggregating MPI parallel components on a node (or on a socket) may actually run slower
• Possible waste of effort
Motivation for Hybrid

- Balance the computational load
- Reduce memory traffic, especially for memory-bound applications
Two Views of a Node

OpenMP

MPI
Two Views = Two Ways to Write Parallel Programs

- OpenMP (or pthreads) only
  - launch one process per node
  - have each process fork one thread (or maybe more) per core
  - share data using shared memory
  - can’t share data with a different process (except maybe via file I/O)

- MPI only
  - launch one process per core, on one node or on many
  - pass messages among processes without concern for location
  - (maybe create different communicators intra-node vs. inter-node)
  - ignore the potential for any memory to be shared

- With hybrid OpenMP/MPI programming, we want each MPI process to launch multiple OpenMP threads that can share local memory
Some Possible MPI + Thread Configurations

- Treat each *node* as an SMP
  - launch a single MPI process per node
  - create parallel threads sharing full-node memory
  - typically want 16 threads/node on Ranger, e.g.
- Treat each *socket* as an SMP
  - launch one MPI process on each socket
  - create parallel threads sharing same-socket memory
  - typically want 4 threads/socket on Ranger, e.g.
- No SMP, ignore shared memory (all MPI)
  - assign an MPI process to each core
  - in a master/worker paradigm, one process per node may be master
  - not really hybrid, may at least make a distinction between nodes
Creating Hybrid Configurations

To achieve configurations like these, we must be able to:

- Assign to each process/thread an *affinity* for some set of cores
- Make sure the *allocation* of memory is appropriately matched
NUMA Operations

Where do processes, threads, and memory allocations get assigned?

• If memory were completely uniform, there would be no need to worry about questions like, “where do processes go?”
• Only for NUMA is the placement of processes/threads and allocated memory (NUMA control) of any importance

The default NUMA control is set through policy

• The policy is applied whenever a process is executed, or a thread is forked, or memory is allocated
• These are all events that are directed from within the kernel

NUMA control is managed by the kernel.
NUMA control can be changed with numactl.
Process Affinity and Memory Policy

- One would like to set the *affinity* of a process for a certain socket or core, and the *allocation* of data in memory relative to a socket or core.
- Individual 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!
- Means by which Process Affinity and Memory Policy can be changed:
  1. dynamically on a running process (knowing process id)
  2. at start of process execution (with wrapper command)
  3. within program through F90/C API
Using numactl, at the Process Level

```
numactl <option socket(s)/core(s)> ./a.out
```

<table>
<thead>
<tr>
<th>For a Process:</th>
<th>For a Process’s Memory:</th>
<th>For a Process:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Socket</strong> Control</td>
<td><strong>Socket</strong> Control</td>
<td><strong>Core</strong> Control</td>
</tr>
<tr>
<td>socket assignment -N</td>
<td>memory allocation -l -i --preferred -m (local, interleaved, preferred, mandatory)</td>
<td>core assignment -C</td>
</tr>
</tbody>
</table>
Quick Guide to `numactl`

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

Hybrid batch script for 16 threads/node

- Make sure **1 process per node** is created
- Specify **total cores allocated** by batch (nodes x 16)
- Set number of **threads for each process**
- PAMPPering at **job level**
  - controls behavior (e.g., process-core affinity) for ALL processes
  - no simple/standard way to control *thread*-core affinity with `numactl`

<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><code>#! -pe 1way 192</code></td>
<td><code>#! -pe 1way 192</code></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><code>export OMP_NUM_THREADS=16</code></td>
<td><code>setenv OMP_NUM_THREADS 16</code></td>
</tr>
<tr>
<td><code>ibrun numactl -i all ./a.out</code></td>
<td><code>ibrun numactl -i all ./a.out</code></td>
</tr>
</tbody>
</table>
SMP Sockets

Hybrid batch script for 4 tasks/node, 4 threads/task

Example script setup for a square (6x6 = 36) processor topology...

- Make sure **4 processes per node** are created (one per socket)
- Specify **total cores allocated** by batch (nodes x 16)
- Specify **actual cores used** with MY_NSLOTS
- Set number of **threads for each process**
- PAMPPering at **process level**, must create script to manage affinity

<table>
<thead>
<tr>
<th>job script (Bourne shell)</th>
<th>job script (C shell)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>#! -pe 4way 48</code></td>
<td><code>#! -pe 4way 48</code></td>
</tr>
<tr>
<td><code>export MY_SLOTS=36</code></td>
<td><code>setenv MY_NSLOTS 36</code></td>
</tr>
<tr>
<td><code>export OMP_NUM_THREADS=4</code></td>
<td><code>setenv OMP_NUM_THREADS 4</code></td>
</tr>
<tr>
<td><code>ibrun numa.sh</code></td>
<td><code>ibrun numa.csh</code></td>
</tr>
</tbody>
</table>
# Script for Socket Affinity

- Example script to extract MPI rank, set numactl options per process
  - on Ranger, MPI ranks are always assigned sequentially, node by node
- Low local ranks → high sockets: tie 0 to socket 3 for best networking

---

<table>
<thead>
<tr>
<th>numa.sh</th>
<th>numa.csh</th>
</tr>
</thead>
</table>
| #!/bin/bash  
export MV2_USE_AFFINITY=0  
export MV2_ENABLE_AFFINITY=0  
#TasksPerNode  
TPN=`echo $PE|sed 's/way//'`  
[ ! $TPN ] && echo TPN null!  
[ ! $TPN ] && exit 1  
#LocalRank, Socket  
LR=$(( $PMI_RANK % $TPN))  
SO=$(( (4*($TPN-$LR))/TPN ))  
numactl -N $SO -m $SO ./a.out |  
#!/bin/csh  
setenv MV2_USE_AFFINITY 0  
setenv MV2_ENABLE_AFFINITY 0  
#TasksPerNode  
set TPN=`echo $PE|sed 's/way//'`  
if(! ${%TPN}) echo TPN null!  
if(! ${%TPN}) exit 1  
#LocalRank, Socket  
@ LR = $PMI_RANK % $TPN  
@ SO = (4*(TPN-$LR))/TPN  
numactl -N $SO -m $SO ./a.out |
Basic Hybrid Program Template

Start with MPI initialization

(Serial regions are executed by the master thread of the MPI process)

Create OMP parallel regions within each MPI process
  – MPI calls may be allowed here too
  – MPI rank is known to all threads

Call MPI in single-threaded regions

Finalize MPI

```
MPI_Init
...
MPI_Call
...
OMP parallel
...
MPI_Call
...
end parallel
...
MPI_Call
...
MPI_Finalize
```
Types of MPI Calls Among Threads

**Single-threaded messaging**
- Call MPI from a serial region
- Call MPI from a single thread within a parallel region

**Multi-threaded messaging**
- Call MPI from multiple threads within a parallel region
- Requires an implementation of MPI that is thread-safe
MPI-2 and Thread Safety

- **Consider thread safety when calling MPI from threads**
- Use MPI_Init_thread to select/determine the level of thread support
  - Supported in MPI-2, substitute for the usual MPI_Init
- Thread safety is identified/controlled by MPI’s provided types
  - *Single* means no multi-threading
  - *Funneled* means only the master thread can call MPI
  - *Serialized* means multiple threads can call MPI, but only 1 call can be in progress at a time
  - *Multiple* means MPI is thread safe
- Monotonic values are assigned to parameters
  - `MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED`
  - `< MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE`
MPI-2’s MPI_Init_thread

Syntax:

```c
int MPI_Init_thread(int &argc, char **&argv, int rqd)
```

- **Input:** `rqd`, or “required” (integer)
  - Indicates the desired level of thread support
- **Output:** `pvd`, or “provided” (integer)
  - Indicates the available level of thread support
- If thread level `rqd` is supported, the call returns `pvd = rqd`
- Otherwise, `pvd` returns the highest provided level of support
# MPI-2 Thread Support Levels

<table>
<thead>
<tr>
<th>Support Levels</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_THREAD_SINGLE</td>
<td>Only one thread will execute.</td>
</tr>
<tr>
<td>MPI_THREAD_FUNNELED</td>
<td>Process may be multi-threaded, but only the main thread will make MPI calls (calls are “funneled” to main thread). <em>Default</em></td>
</tr>
<tr>
<td>MPI_THREAD_SERIALIZE</td>
<td>Process may be multi-threaded, and any thread can make MPI calls, but threads cannot execute MPI calls concurrently; they must take turns (calls are “serialized”).</td>
</tr>
<tr>
<td>MPI_THREAD_MULTIPLE</td>
<td>Multiple threads may call MPI, with no restriction.</td>
</tr>
</tbody>
</table>
## Example: Single-Threaded MPI Calls

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>include 'mpif.h'</code></td>
<td><code>#include &lt;mpi.h&gt;</code></td>
</tr>
<tr>
<td><code>program hybsimp</code></td>
<td><code>int main(int argc,</code></td>
</tr>
<tr>
<td></td>
<td><code>    char **argv) {</code></td>
</tr>
<tr>
<td><code>call MPI_Init(ie)</code></td>
<td><code>    int rank, size, ie, i;</code></td>
</tr>
<tr>
<td><code>call MPI_Comm_rank(...irk,ie)</code></td>
<td><code>    ie= MPI_Init(&amp;argc,&amp;argv[]);</code></td>
</tr>
<tr>
<td><code>call MPI_Comm_size(...isz,ie)</code></td>
<td><code>    ie= MPI_Comm_rank(...&amp;rank);</code></td>
</tr>
<tr>
<td><code>!Setup shared mem, comp/comm</code></td>
<td><code>    ie= MPI_Comm_size(...&amp;size);</code></td>
</tr>
<tr>
<td><code>!$OMP parallel do</code></td>
<td><code>#pragma omp parallel for</code></td>
</tr>
<tr>
<td><code>    do i=1,n</code></td>
<td><code>for(i=0; i&lt;n; i++){</code></td>
</tr>
<tr>
<td><code>      &lt;work&gt;</code></td>
<td><code>         &lt;work&gt;</code></td>
</tr>
<tr>
<td><code>    enddo</code></td>
<td><code>    }</code></td>
</tr>
<tr>
<td><code>!Compute &amp; communicate</code></td>
<td><code>// compute &amp; communicate</code></td>
</tr>
<tr>
<td><code>call MPI_Finalize(ierr)</code></td>
<td><code>ie= MPI_Finalize();</code></td>
</tr>
<tr>
<td><code>end</code></td>
<td>}</td>
</tr>
</tbody>
</table>
Funneled MPI Calls via Master

• Must have support for MPI_THREAD_FUNNELEDED or higher
• Best to use OMP_BARRIER
  – there is no implicit barrier in the master workshare construct, OMP_MASTER
  – in the example, the master thread will execute a single MPI call within the OMP_MASTER construct
  – all other threads will be sleeping
Example: Funneled MPI Calls via Master

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>include 'mpif.h'</code></td>
<td></td>
</tr>
<tr>
<td><code>program hybmas</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>!$OMP parallel</code></td>
<td></td>
</tr>
<tr>
<td><code>!$OMP barrier</code></td>
<td></td>
</tr>
<tr>
<td><code>!$OMP master</code></td>
<td></td>
</tr>
<tr>
<td><code>call MPI_&lt;Whatever&gt;(...,ie)</code></td>
<td></td>
</tr>
<tr>
<td><code>!$OMP end master</code></td>
<td></td>
</tr>
<tr>
<td><code>!$OMP barrier</code></td>
<td></td>
</tr>
<tr>
<td><code>!$OMP end parallel</code></td>
<td></td>
</tr>
<tr>
<td><code>end</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>#include &lt;mpi.h&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>int main(int argc,</code></td>
<td></td>
</tr>
<tr>
<td><code>    char **argv) {</code></td>
<td></td>
</tr>
<tr>
<td><code>    int rank, size, ie, i;</code></td>
<td></td>
</tr>
<tr>
<td><code>    #pragma omp parallel</code></td>
<td></td>
</tr>
<tr>
<td><code>{</code></td>
<td></td>
</tr>
<tr>
<td><code>    #pragma omp barrier</code></td>
<td></td>
</tr>
<tr>
<td><code>    #pragma omp master</code></td>
<td></td>
</tr>
<tr>
<td><code>{</code></td>
<td></td>
</tr>
<tr>
<td><code>        ie= MPI_&lt;Whatever&gt;(...);</code></td>
<td></td>
</tr>
<tr>
<td><code>    }</code></td>
<td></td>
</tr>
<tr>
<td><code>    #pragma omp barrier</code></td>
<td></td>
</tr>
<tr>
<td><code>{</code></td>
<td></td>
</tr>
<tr>
<td><code>    }</code></td>
<td></td>
</tr>
</tbody>
</table>
Serialized MPI Calls and OpenMP

- Must have support for `MPI_THREAD_SERIALIZED` or higher
- Best to use `OMP_BARRIER` only at beginning, since there is an implicit barrier in the SINGLE workshare construct, OMP_SINGLE
  - Example is the simplest one: any thread (not necessarily master) will execute a single MPI call within the OMP_SINGLE construct
  - All other threads will be sleeping
### Example: Serialized MPI Calls and OpenMP

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>include 'mpif.h'</code></td>
<td><code>#include &lt;mpi.h&gt;</code></td>
</tr>
<tr>
<td><code>program hybsing</code></td>
<td><code>int main(int argc,</code></td>
</tr>
<tr>
<td><code>call MPI_Init_thread( &amp; </code></td>
<td><code>   char **argv) {</code></td>
</tr>
<tr>
<td><code>MPI_THREAD_SERIALIZED,ipvd,ie)</code></td>
<td><code>   int rank, size, ie, i;</code></td>
</tr>
<tr>
<td><code>!$OMP parallel</code></td>
<td><code>   ie= MPI_Init_thread(</code></td>
</tr>
<tr>
<td><code>!$OMP barrier</code></td>
<td><code>   MPI_THREAD_SERIALIZED,ipvd));</code></td>
</tr>
<tr>
<td><code>!$OMP single</code></td>
<td><code>   #pragma omp parallel {</code></td>
</tr>
<tr>
<td><code>call MPI_&lt;Whatever&gt;(...,ie)</code></td>
<td><code>   #pragma omp barrier</code></td>
</tr>
<tr>
<td><code>!$OMP end single</code></td>
<td><code>   #pragma omp master</code></td>
</tr>
<tr>
<td><code>!Don't need OMP barrier</code></td>
<td><code>   {</code></td>
</tr>
<tr>
<td><code>!$OMP end parallel</code></td>
<td><code>     ie= MPI_&lt;Whatever&gt;(...);</code></td>
</tr>
<tr>
<td><code>end</code></td>
<td><code>} //Don't need omp barrier</code></td>
</tr>
<tr>
<td></td>
<td>`}</td>
</tr>
</tbody>
</table>
Overlapping Work & MPI Calls

• One core is capable of saturating the lanes of the PCIe network link...
  – Why use all cores to communicate?
  – Instead, communicate using just one or several cores
  – Can do work with the rest during communication

• Must have support for MPI_THREAD_FUNNELED or higher to do this

• Can be difficult to manage and load-balance!
Example: Overlapping Work & MPI Calls

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>include 'mpif.h'</td>
<td>#include &lt;mpi.h&gt;</td>
</tr>
<tr>
<td>program hybsing</td>
<td>int main(int argc,</td>
</tr>
<tr>
<td></td>
<td>char **argv) {</td>
</tr>
<tr>
<td>!$OMP parallel</td>
<td>int rank, size, ie, i;</td>
</tr>
<tr>
<td>if (ithread .eq. 0) then</td>
<td>#pragma omp parallel</td>
</tr>
<tr>
<td>call MPI_&lt;Whatever&gt;(...,ie)</td>
<td>{</td>
</tr>
<tr>
<td>else</td>
<td>if (thread == 0){</td>
</tr>
<tr>
<td>&lt;work&gt;</td>
<td>ie= MPI_&lt;Whatever&gt;(...);</td>
</tr>
<tr>
<td>endif</td>
<td>}</td>
</tr>
<tr>
<td>!$OMP end parallel</td>
<td>if(thread != 0){</td>
</tr>
<tr>
<td>end</td>
<td>&lt;work&gt;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
Multiple Threads Calling MPI

• Thread ID as well as rank can be used in communication
• Technique is illustrated in multi-thread “ping” (send/receive) example
Example: Multiple Threads Calling MPI

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)

...!

!$OMP parallel private(j, ithread, nthreads)
    nthreads=OMP_GET_NUM_THREADS()
    ithread =OMP_GET_THREAD_NUM()
call pwork(ithread, irank, nthreads, nranks...)
if(irank == 0) then
    call mpi_send(ithread,1,MPI_INTEGER, 1, ithread, MPI_COMM_WORLD, ierr)
else
    call mpi_recv( j,1,MPI_INTEGER, 0, ithread, MPI_COMM_WORLD, istat, ierr)
    print*, "Yep, this is ",irank," thread ", ithread," I received from ", j
endif
!$OMP END PARALLEL
end
NUMA Control in Code, at the Thread Level

- Within a code, **Scheduling Affinity** and **Memory Policy** can be examined and changed through:
  - sched_getaffinity, sched_setaffinity
  - get_mempolicy, set_mempolicy
- This is the *only* way to set affinities and policies that differ per *thread*
- To make scheduling assignments, set bits in a mask:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
Assignment to Core 0

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Assignment to Core 15

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
Assignment to Core 0 or 15
```
...  
#include <spawn.h>  //C API parameters and prototypes
...

int icore=3;  //Set core number
CPU_ZERO( &cpu_mask);  //Set mask to zero
CPU_SET(icore,&cpu_mask);  //Set mask with core 

err = sched_setaffinity( (pid_t)0 ,  //Set the affinity
sizeof(cpu_mask),
&cpu_mask);
Conclusions and Future Prospects

- On NUMA systems like Ranger, placement and binding of processes and their associated memory are important performance considerations.
- Process Affinity and Memory Policy have a significant effect on pure MPI, pure OpenMP, and Hybrid codes.
- Simple numactl commands and APIs allow users to control affinity of processes and threads and memory assignments.
- Future prospects for hybrid programming:
  - 8-core and 16-core socket systems are on the way, so even more effort will be focused on process scheduling and data locality.
  - Expect to see more multi-threaded libraries; be alert for their potential interaction with your own multithreading strategy.
References

- Yun (Helen) He and Chris Ding, Lawrence Berkeley National Laboratory, June 24, 2004: [Hybrid OpenMP and MPI Programming and Tuning (NUG2004)](www.nersc.gov/nusers/services/training/classes/NUG/Jun04/NUG2004_yhe_hybrid.ppt).

- Texas Advanced Computing Center: [Ranger User Guide](www.tacc.utexas.edu/services/userguides/ranger), see numa section.
