Multi-Core Optimizations
Memory & Process/Thread Control

Kent Milfeld

Optimization
NUMA
August 2, 2010
Outline

• Memory View and Modes of Parallel Processing
• NUMA Controls—what are they good for
  – Purpose
  – Defaults
  – Example Cases: Memory and Process Location
• Hybrid Computing
  – Simple “how to”
• GPUs
  – Basics
  – Memory Considerations
Memory Hierarchy

• Parallel Paradigms
  – MPI: addresses data movement in distributed memory (between processes-- executables)
  – OpenMP: addresses data access in shared memory (among threads in an executable)
• Processes/Threads and Data on Nodes
  – Hardware: Memory Access makes a difference
  – Control: It’s about location, location, location.
Node Views

Paradigms

OpenMP

- Run a bunch of threads in shared memory (spawned by a single a.out).

MPI

- Run a bunch of a.out’s as distributed memory paradigm

Hardware

Ranger Node

Node: **Non-Uniform Memory Access (NUMA)**

- 4 sockets each with 4 cores

Reality of Process Location & Data Storage
Modes of MPI/OpenMP Operation

- Pure MPI Node
  - 16 MPI Processes
  - 4 MPI Processes
  - 1 MPI Process

- Pure SMP Node
  - 16 Threads/Process
  - 4 Threads/Process
  - 0 Threads/Process

Master Thread of MPI Task
- MPI Process on Core (a.out)
- Master Process-Thread of MPI Task
- Spawned Thread of MPI Task
- Memory of CPU
What are Numa controls?

• **Process Affinity**
  – Positions process or thread on a core
    • Process and threads are treated equally
    • User request to kernel
    • May be overridden by kernel Scheduling

• **Memory Policy**
  – Specifies location of data storage
    • Applies to process/threads (children inherit policy)
    • User request to kernel overrides default policy
Why use Numa Controls?

• Why Process Affinity
  – On a laptop there may be ~500 threads and numerous applications running. Application time slicing is optimal served with relocation.
  – On an HPC node, it is better to anchor an application’s process or thread onto one core.

• Why Memory Policy
  – On a laptop memory access is uniform (1 CPU with multiple cores)
  – HPC nodes have Non Uniform Memory Access because they are multi-CPU servers.
Process/Thread Concerns

• Process Affinity
  – A Process/Thread (P/T) is scheduled by kernel.
  – Moving P/Ts to other CPUs often forces remote memory access by the moved P/T.
  – It is necessary to distribute processes when the process/thread count < core count on a node (for large-memory/core jobs).
  – Even MPI latency is affected by location.
Numa defaults

• Memory Policy
  – The Physical Memory (of a socket) is assigned to code data in pages (4K Bytes, default)
  – On First Access, data assigned to physical (socket) memory of process or thread of execution (local, default)
  – Once assigned, data does not move (Standard Linux). (Some system have “page migration”.)
  – If local memory is full, spill over to non-local (default behavior)
3 Important Concepts

- Virtual Memory (code data) is assigned in Physical Memory Pages (4KB)
- Code Data (anonymous pages) is user controlled
- Cache Buffers (for IO) is controlled by the kernel.
## Modes of MPI/OpenMP Operation

<table>
<thead>
<tr>
<th>Process (a.out) Location</th>
<th>Use defaults (no concern)</th>
<th>Force 4 a.out’s across sockets</th>
<th>Force thread-level affinity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage Access</td>
<td>FORCE* local</td>
<td>FORCE* local</td>
<td>Application dependent</td>
</tr>
</tbody>
</table>

Suggested modes, see [tacc affinity](#) script.
Case Study
(Problem: Variable Performance)

• After a maintenance, some codes would run their fastest for a few days, and then subsequent job times would sometimes increase significantly by varied amounts.
Case Study
(Observations)

• Jobs that perform I/O use system “buffers” for caching.

• As long as files persist, the system tries to keep the buffer blocks in memory—EVEN ACROSS JOBS.

• Even worse, if IO is performed from rank 0 (as is often the case) the job will leave buffer blocks on the memory of a single CPU (on Ranger).
Case Study
(Demonstrating IO buffer Impact on Applications)

• Example (worst case scenario on Ranger)
  – Writing an 8GB file from CPU 0 leaves 6GB of buffer on CPU 1’s memory.
  – A subsequent DAXPY operation (requiring 7GB) on sockets 1 will run \(~50\%\) slower (e.g. 2.2 s instead of 1.4 for test case).

• On TACC systems, avoid this by using `tacc_affinity`. It forces IO buffers to be released. Batch launch:
  `ibrunch tacc_affinity ./my_executable`
First Job (large IO, moderate memory size)
- writes large IO file(s) from socket 0.
- IO buffers are consumed on Memory 1 until memory is nearly full, and then spills over to other memory units.
- IO buffers persist in memory (red)
- Once full, memory assignments from socket 1 spill over to non-local memory on socket 2 and 3.

Second Job (moderate memory size)
- Memory requests from socket 1.
- Memory assignments spilled over to remote memories.
Thread/Process Location

- **Maintain Memory Locality.** A process/thread should retain possession of the core where it initializes its data.
- **MPI messaging is directed through 1 socket.**
- **Daemons have preset affinity for a socket and even use one particular core (0) and localized memory.**
Node Asymmetry

TACC Ranger Node
MPI Latency

– MPI Bandwidth is not appreciably affected by socket location of process. Memory bandwidth is much higher than network bandwidth.

– Latency can be affected by location on some systems.
   • Very small effect from socket location
     – Ranger: Latency ~3.0us (~3% variation)
   • Possibly affected by core use
     Contention with system thread & daemon assignments
     – TACC Longhorn: Latency 2.60us on core 0, 1.85us others
     – TACC Ranger: no significant variation
NUMA Operations

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

• Users can alter Kernel Policies (setting Process Affinity and Memory Policy == PAMPPer)
  – Users can PAMPPer their own processes.
  – Root can PAMPPer any process.
  – Careful, libraries may PAMPPer, too!
NUMA Control at Launch

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

**Command:**

```
numactl < options socket/core > ./a.out
```

**Diagram:**

- **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`
# numactl Quick Guide

<table>
<thead>
<tr>
<th>Affinity Type</th>
<th>Command</th>
<th>Option</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Socket Affinity</strong></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><strong>Memory Policy</strong></td>
<td>numactl</td>
<td>-l</td>
<td>{no argument}</td>
<td>Allocate on current socket. Fallback to any other if full.</td>
</tr>
<tr>
<td></td>
<td>numactl</td>
<td>-i</td>
<td>{0,1,2,3}</td>
<td>Allocate round robin (interleave) on these sockets. No fallback.</td>
</tr>
<tr>
<td></td>
<td>numactl</td>
<td>--preferred=</td>
<td>{0,1,2,3} select only one</td>
<td>Allocate on this socket; fallback to any other if full.</td>
</tr>
<tr>
<td></td>
<td>numactl</td>
<td>-m</td>
<td>{0,1,2,3}</td>
<td>Only allocate on this (these) socket(s). No fallback.</td>
</tr>
<tr>
<td><strong>Core Affinity</strong></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>
NUMA Control in Code

• **Scheduling Affinity** and **Memory Policy** can be changed within code with API:
  
  sched_get/setaffinity
  get/set_memorypolicy

• Scheduling: Bits in a mask are set for assignments.

Assignment to Core 0

Assignment to Core 15

Assignment to Core 0 or 15
NUMA in Code

• **Scheduling Affinity**

```c
#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
NUMA in Code

- **set_mempolicy**

```c
#include <numaif.h>
...
unsigned long maxnode=4, nodemask=0;
int mode;
...
mode=MPOL_BIND;
nodemask=((1<<inode));

iret = set_mempolicy( mode, 
                       &nodemask, 
                       maxnode);
```

### Policies (mode)
- MPOL_DEFAULT
- MPOL_PREFERRED
- MPOL_BIND
- MPOL_INTERLEAVE

### C API params/protos
- Set max sockets, allocate mask, mode var.
- Set mode
- Set mask to socket (inode=socket #)
  - Set policy
Hybrid - Motivation

- Load Balancing
- Reduce Memory Traffic
• If thread access is mainly local, each thread should initialize its portion of data. Do NOT initialize in serial section of code.
• If each thread accesses all of memory, try interleave memory policy.

Remote Access: 1.2-1.8 x slower.
Local Access: Bandwidth is highest.
Hybrid – Program Model

• Start with Special 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 or parallel regions.
• Finalize MPI
MPI with OpenMP -- Messaging

Multi-threaded:
MPI through Master
(may not be in parallel region)

Multi-threaded:
MPI with all/any thread

MPI from serial region or a single thread within parallel region

rank to rank

MPI from multiple threads within parallel region
Requires thread-safe implementation
Hybrid Coding – MPI with Master

Fortran

```fortran
include 'mpif.h'
use omp_lib
program hybsimp

call MPI_Init(ierr)
call MPI_Comm_rank (... , irank, ierr)
call MPI_Comm_size (... , isize, ierr)
!
  MPI with Master thread
!
$OMP parallel
do i=1,n
  <work>
enddo
!
  MPI with Master thread

! ierr= MPI_Finalize(ierr)
end
```

C

```c
#include <mpi.h>
#include <omp.h>
int main(int argc, char **argv){

ierr= MPI_Init(&argc,&argv[0]);
ierr= MPI_Comm_rank (... , &rank);
ierr= MPI_Comm_size (... , &size);

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

// MPI with Master

ierr= MPI_Finalize();
}
```
Hybrid Coding – MPI with threads

```fortran
#include 'mpif.h'
use omp_lib
program hybsing
  call MPI_Init_thread(MPI_THREAD_MULTIPLE, iprovided, ierr)
!	MPI with Master thread
 !$OMP parallel
!	OMPI barrier
!	call MPI_<Whatever>(..., ierr) {any/all threads}
!	OMPI end parallel
!	MPI with Master thread
end
```

```c
#include <mpi.h>
#include <omp.h>
int main(int argc, char **argv){
  MPI_Init_thread(..., MPI_THREAD_MULTIPLE, iprovided)
  //MPI with Master
  #pragma omp parallel
  {
    #pragma omp barrier
    ierr=MPI_<Whatever>(...) {any/all threads}
  }
  //MPI with Master
}
```
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
GPU Considerations

- Programming Model/Architecture
- Memory Access
- Threading
What is a GPU?

• GPU = graphics processing unit
• High-end GPUs can now do threaded “SIMD” floating point operations on 64-bit data.

This GPU is specifically designed for HPC Computing – it has no graphics output.
The GPU programming model

Runtime Environment:

- CPU Host (Process or Threads) CPU & GPU code compiled here.
- From within CPU code, GPU routines (kernels) are launched & CPU waits.
- Data can be moved between CPU and GPU by either GPU or CPU.

TACC Longhorn Node
2-CPUs/2GPUs
The GPU Memory

16K for registers: used by all threads in SM - no context switching
16K shared mem is shared among threads in a block

Global mem is fast, but slow compared to shared. Use Shared like a cache.
CPU/GPU thread paradigms

- CPU, OpenMP→threads
- GPU, blocks & threads

```
loop i=1->M
loop j=1->N
loop l=1->K
  sum=sum+a(i,l)*b(l,j)
end;
  c(i,j)=sum
end;end
```

Course-grain parallelism
Asynchronous work-sharing groups

```
Blocks
Multiple Threads
  sum=sum+lba(it)*lbb(it)
end;end
```

it = thread index
lba/lbb = local blocks in Shared

Fine-grain parallelism
Lock-step execution in WARP *
* 32 threads SIMT
Single Instruction Multiple Thread
Conclusion

- Placement and binding of processes, and allocation location of memory are important performance considerations in pure MPI/OpenMP, Hybrid codes and GPUs.
- Simple `numactl` commands and APIs allow users to control process and memory assignments.
- For 8/12/16-core systems with multiple sockets, even more effort will be focused on process scheduling and memory location. GPUs employ 10-20x more cores.
- Expect to see more multi-threaded libraries.
References

- [www.nersc.gov/nusers/services/training/classes/NUG/Jun04/NUG2004_yhe_hybrid.ppt](www.nersc.gov/nusers/services/training/classes/NUG/Jun04/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.tacc.utexas.edu/services/userguides/ranger](www.tacc.utexas.edu/services/userguides/ranger) {See numa section.}

- [www.mpi-forum.org/docs/mpi2-report.pdf](www.mpi-forum.org/docs/mpi2-report.pdf)

- [www.nersc.gov/nusers/services/training/classes/NUG/Jun04/NUG2004_yhe_hybrid.ppt](www.nersc.gov/nusers/services/training/classes/NUG/Jun04/NUG2004_yhe_hybrid.ppt)