Hybrid MPI and OpenMP Parallel Programming

MPI + OpenMP and other models on clusters of SMP nodes

Rolf Rabenseifner\textsuperscript{1)} \quad Georg Hager\textsuperscript{2)} \quad Gabriele Jost\textsuperscript{3)}

\textsuperscript{1)} High Performance Computing Center (HLRS), University of Stuttgart, Germany
\textsuperscript{2)} Regional Computing Center (RRZE), University of Erlangen, Germany
\textsuperscript{3)} Texas Advanced Computing Center / Naval Postgraduate School, USA

Tutorial M09 at SUPERCOMPUTING 2009 (SC09)
Portland, Oregon, USA, Nov. 16
Outline

1. Introduction / Motivation 2
2. Programming models on clusters of SMP nodes 6
3. Case Studies / pure MPI vs hybrid MPI+OpenMP 13
4. Practical “How-To” on hybrid programming 43
5. Mismatch Problems 80
6. Opportunities: Application categories that can benefit from hybrid parallelization 103
7. Thread-safety quality of MPI libraries 112
8. Tools for debugging and profiling MPI+OpenMP 119
9. Summary 126
10. Appendix 134
11. Content (detailed) 149

Slide number

8:30 – 10:00

10:30 – 12:00
Motivation

- Efficient programming of clusters of SMP nodes
  - SMP nodes:
    - Dual/multi core CPUs
    - Multi CPU shared memory
    - Multi CPU ccNUMA
    - Any mixture with shared memory programming model
  - Hardware range
    - mini-cluster with dual-core CPUs
    - ...
    - large constellations with large SMP nodes
      - ... with several sockets (CPUs) per SMP node
      - ... with several cores per socket
      - Hierarchical system layout
  - Hybrid MPI/OpenMP programming seems natural
    - MPI between the nodes
    - OpenMP inside of each SMP node
Motivation

- Which programming model is fastest?
- MPI everywhere?
- Fully hybrid MPI & OpenMP?
- Something between? (Mixed model)
- Often hybrid programming slower than pure MPI
  - Examples, Reasons, …
Goals of this tutorial

- Sensitize to problems on clusters of SMP nodes
  - see sections → Case studies → Mismatch problems
- Technical aspects of hybrid programming
  - see sections → Programming models on clusters → Examples on hybrid programming
- Opportunities with hybrid programming
  - see section → Opportunities: Application categories that can benefit from hybrid paralleliz.
- Issues and their Solutions
  - with sections → Thread-safety quality of MPI libraries → Tools for debugging and profiling for MPI+OpenMP

• Less frustration &
• More success with your parallel program on clusters of SMP nodes
Outline

- Introduction / Motivation
- Programming models on clusters of SMP nodes
  - Case Studies / pure MPI vs hybrid MPI+OpenMP
  - Practical “How-To” on hybrid programming
  - Mismatch Problems
  - Opportunities:
    - Application categories that can benefit from hybrid parallelization
  - Thread-safety quality of MPI libraries
  - Tools for debugging and profiling MPI+OpenMP
  - Summary
Major Programming models on hybrid systems

- Pure MPI (one MPI process on each core)
- Hybrid MPI+OpenMP
  - shared memory OpenMP
  - distributed memory MPI

- Other: Virtual shared memory systems, PGAS, HPF, …
- Often **hybrid programming (MPI+OpenMP)** slower than **pure MPI**
  - why?

### MPI

Sequential program on each core

**Explicit Message Passing** by calling *MPI_Send & MPI_Recv*

### OpenMP

(shared data)

```c
some_serial_code
#pragma omp parallel for
for (j=...;...; j++)
    block_to_be_parallelized
again_some_serial_code
```

- *Master thread, other threads*
  - ***sleeping***
Parallel Programming Models on Hybrid Platforms

- **pure MPI**
  - one MPI process on each core

- **hybrid MPI+OpenMP**
  - MPI: inter-node communication
  - OpenMP: inside of each SMP node

- **OpenMP only**
  - distributed virtual shared memory

**No overlap of Comm. + Comp.**
- MPI only outside of parallel regions of the numerical application code

**Overlapping Comm. + Comp.**
- MPI communication by one or a few threads while other threads are computing

**Masteronly**
- MPI only outside of parallel regions
Pure MPI

Advantages
- No modifications on existing MPI codes
- MPI library need not to support multiple threads

Major problems
- Does MPI library uses internally different protocols?
  - Shared memory inside of the SMP nodes
  - Network communication between the nodes
- Does application topology fit on hardware topology?
- Unnecessary MPI-communication inside of SMP nodes!

Discussed in detail later on in the section Mismatch Problems
Hybrid Masteronly

Advantages
- No message passing inside of the SMP nodes
- No topology problem

Major Problems
- All other threads are sleeping while master thread communicates!
- Which inter-node bandwidth?
- MPI-lib must support at least MPI_THREAD_FUNNELED

for (iteration ....) {
    #pragma omp parallel
    numerical code
    /*end omp parallel */
    /* on master thread only */
    MPI_Send (original data to halo areas in other SMP nodes)
    MPI_Recv (halo data from the neighbors)
} /*end for loop*/

→ Section Thread-safety quality of MPI libraries
if (my_thread_rank < ...) {
    MPI_Send/Recv....
    i.e., communicate all halo data
} else {
    Execute those parts of the application
    that do not need halo data
    (on non-communicating threads)
}

Execute those parts of the application
that need halo data
(on all threads)
Pure OpenMP (on the cluster)

- Distributed shared virtual memory system needed
- Must support clusters of SMP nodes
- e.g., Intel® Cluster OpenMP
  - Shared memory parallel inside of SMP nodes
  - Communication of modified parts of pages at OpenMP flush (part of each OpenMP barrier)

Experience:
- Mismatch section

i.e., the OpenMP memory and parallelization model is prepared for clusters!
Outline

• Introduction / Motivation
• Programming models on clusters of SMP nodes

• Case Studies / pure MPI vs hybrid MPI+OpenMP
  – The Multi-Zone NAS Parallel Benchmarks
  – For each application we discuss:
    • Benchmark implementations based on different strategies and programming paradigms
    • Performance results and analysis on different hardware architectures
  – Compilation and Execution Summary

Gabriele Jost (University of Texas, TACC/Naval Postgraduate School, Monterey CA)

• Practical “How-To” on hybrid programming
• Mismatch Problems
• Opportunities: Application categories that can benefit from hybrid parallelism.
• Thread-safety quality of MPI libraries
• Tools for debugging and profiling MPI+OpenMP
• Summary
### The Multi-Zone NAS Parallel Benchmarks

- Multi-zone versions of the NAS Parallel Benchmarks LU, SP, and BT
- Two hybrid sample implementations
- Load balance heuristics part of sample codes
- [www.nas.nasa.gov/Resources/Software/software.html](http://www.nas.nasa.gov/Resources/Software/software.html)

<table>
<thead>
<tr>
<th></th>
<th>MPI/OpenMP</th>
<th>MLP</th>
<th>Nested OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time step</td>
<td>sequential</td>
<td>sequential</td>
<td>sequential</td>
</tr>
<tr>
<td>inter-zones</td>
<td>MPI Processes</td>
<td>MLP Processes</td>
<td>OpenMP</td>
</tr>
<tr>
<td>exchange boundaries</td>
<td>Call MPI</td>
<td>data copy+ sync.</td>
<td>OpenMP</td>
</tr>
<tr>
<td>intra-zones</td>
<td>OpenMP</td>
<td>OpenMP</td>
<td>OpenMP</td>
</tr>
</tbody>
</table>
Using MPI/OpenMP

call omp_set_numthreads (weight)
do step = 1, itmax
   call exch_qbc(u, qbc, nx, ...)
call mpi_send/recv

    call mpi_send/recv

do zone = 1, num_zones
   if (iam .eq. pzone_id(zone)) then
      call ssor(u,rsd,...)
      end if
   end do
end do
...
Benchmark Characteristics

- **Aggregate sizes:**
  - Class B: 304 x 208 x 17 grid points
  - Class C: 480 x 320 x 28 grid points
  - Class D: 1632 x 1216 x 34 grid points
  - Class E: 4224 x 3456 x 92 grid points

- **BT-MZ:** (Block tridiagonal simulated CFD application)
  - #Zones: 64 (Class B), 256 (C), 1024 (D), 4096 (E)
  - Size of the zones varies widely:
    - large/small about 20
    - requires multi-level parallelism to achieve a good load-balance

- **LU-MZ:** (LU decomposition simulated CFD application)
  - #Zones: 16 (Class B, C, and D)
  - Size of the zones identical:
    - no load-balancing required
    - limited parallelism on outer level

- **SP-MZ:** (Scalar Pentadiagonal simulated CFD application)
  - #Zones: 64 (Class B), 256 (C), 1024 (D), 4096 (E)
  - Size of zones identical
    - no load-balancing required

**Expectations:**
- Pure MPI: Load-balancing problems!
- Good candidate for MPI+OpenMP
- Limited MPI Parallelism: MPI+OpenMP increases Parallelism
- Load-balanced on MPI level: Pure MPI should perform best
Benchmark Architectures

- Sun Constellation (Ranger)
- Cray XT5
- Cray XT4 (skipped, i.e., only in the handouts)
- IBM Power 6
Hybrid code on cc-NUMA architectures

- **OpenMP:**
  - Support only per MPI process
  - Version 2.5 does not provide support to control to map threads onto CPUs. Support to specify thread affinities was under discussion for 3.0 but has not been included
- **MPI:**
  - Initially not designed for NUMA architectures or mixing of threads and processes, MPI-2 supports threads in MPI
  - API does not provide support for memory/thread placement
- **Vendor specific APIs to control thread and memory placement:**
  - Environment variables
  - System commands like `numactl`
Sun Constellation Cluster Ranger (1)

- Located at the Texas Advanced Computing Center (TACC), University of Texas at Austin (http://www.tacc.utexas.edu)
- 3936 Sun Blades, 4 AMD Quad-core 64bit 2.3GHz processors per node (blade), 62976 cores total
- 123TB aggregate memory
- Peak Performance 579 Tflops
- InfiniBand Switch interconnect
- Sun Blade x6420 Compute Node:
  - 4 Sockets per node
  - 4 cores per socket
  - HyperTransport System Bus
  - 32GB memory
Sun Constellation Cluster Ranger (2)

- **Compilation:**
  - PGI pgf90 7.1
  - mpif90 -tp barcelona-64 -r8

- **Cache optimized benchmarks**
  - Execution:
    - MPI MVAPICH
    - setenv OMP_NUM_THREAD NTHREAD
    - Ibrun numactl bt-mz.exe

- **numactl** controls
  - Socket affinity: select sockets to run
  - Core affinity: select cores within socket
  - Memory policy: where to allocate memory
Hybrid Parallel Programming
Rabenseifner, Hager, Jost

SUN: NPB-MZ Class E Scalability on Ranger

- Scalability in Mflops
- MPI/OpenMP outperforms pure MPI
- Use of numactl essential to achieve scalability

NPB-MZ Class E Scalability on Sun Constellation

- SP-MZ (MPI)
- SP-MZ MPI+OpenMP
- BT-MZ (MPI)
- BT-MZ MPI+OpenMP

**BT**
Significant improvement (235%):
Load-balancing issues solved with MPI+OpenMP

**SP**
Pure MPI is already load-balanced.
But hybrid programming 9.6% faster

Cannot be build for 8192 processes!

Hybrid:
**SP:** still scales
**BT:** does not scale
SUN: Running hybrid on Sun Constellation Cluster Ranger

- Highly hierarchical
- Shared Memory:
  - Cache-coherent, Non-uniform memory access (ccNUMA) 16-way Node (Blade)
- Distributed memory:
  - Network of ccNUMA blades
    - Core-to-Core
    - Socket-to-Socket
    - Blade-to-Blade
    - Chassis-to-Chassis
MPI ping-pong micro benchmark results on Ranger

- Inside one node:
  
  Ping-pong socket 0 with 1, 2, 3 and 1, 2, or 4 simultaneous comm. (quad-core)
  
  - Missing Connection: Communication between socket 0 and 3 is slower
  
  - Maximum bandwidth: 1 x 1180, 2 x 730, 4 x 300 MB/s

- Node-to-node inside one chassis with 1-6 node-pairs (= 2-12 procs)
  
  - Perfect scaling for up to 6 simultaneous communications
  
  - Max. bandwidth: 6 x 900 MB/s

- Chassis to chassis (distance: 7 hops) with 1 MPI process per node and 1-12 simultaneous communication links
  
  - Max: 2 x 900 up to 12 x 450 MB/s

“Exploiting Multi-Level Parallelism on the Sun Constellation System”, L. Koesterke, et al., TACC, TeraGrid08 Paper
NUMA Control: Process Placement

- Affinity and Policy can be changed externally through `numactl` at the socket and core level.

```
Command:  numactl  <options>  ja.out
```

```
2
Core  Core
Core  Core
Core  Core
Core  Core

3
Core  Core
Core  Core
Core  Core
Core  Core

8,9,10,11
Core  Core
Core  Core
Core  Core
Core  Core

12,13,14,15
Core  Core
Core  Core
Core  Core
Core  Core

1
Core  Core
Core  Core
Core  Core
Core  Core

0
Core  Core
Core  Core
Core  Core
Core  Core

4,5,6,7
Core  Core
Core  Core
Core  Core
Core  Core

0,1,2,3
Core  Core
Core  Core
Core  Core
Core  Core
```
NUMA Operations: Memory Placement

Memory allocation:
- MPI
  - local allocation is best
- OpenMP
  - Interleave best for large, completely shared arrays that are randomly accessed by different threads
  - local best for private arrays
- Once allocated, a memory-structure is fixed
<table>
<thead>
<tr>
<th></th>
<th>cmd</th>
<th>option</th>
<th>arguments</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket Affinity</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>Memory Policy</td>
<td>numactl</td>
<td>-l</td>
<td>{no argument}</td>
<td>Allocate on current socket.</td>
</tr>
<tr>
<td>Memory Policy</td>
<td>numactl</td>
<td>-i</td>
<td>{0,1,2,3}</td>
<td>Allocate round robin (interleave) on these sockets.</td>
</tr>
<tr>
<td>Memory Policy</td>
<td>numactl</td>
<td>--preferred=</td>
<td>{0,1,2,3}</td>
<td>Allocate on this socket; fallback to any other if full.</td>
</tr>
<tr>
<td>Memory Policy</td>
<td>numactl</td>
<td>-m</td>
<td>{0,1,2,3}</td>
<td>Only allocate on this (these) socket(s).</td>
</tr>
<tr>
<td>Core Affinity</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>
### Hybrid Batch Script: 4 tasks, 4 threads/task

<table>
<thead>
<tr>
<th>job script</th>
<th>job script</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(Bourne shell)</strong></td>
<td><strong>(C shell)</strong></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><code>#! -pe 4way 32</code></td>
<td><code>#! -pe 4way 32</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>

#### numa.sh

```bash
#!/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

```tcsh
#!/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
```
Numactl – Pitfalls: Using Threads across Sockets

bt-mz.1024x8 yields best load-balance

- pe 2way 8192
- export OMP_NUM_THREADS=8

my_rank=$PMI_RANK
local_rank=$(( $my_rank % $myway ))
numnode=$(( $local_rank + 1 ))

Original:
numactl -N $numnode -m $numnode *

Bad performance!
- Each process runs 8 threads on 4 cores
- Memory allocated on one socket
Numactl – Pitfalls: Using Threads across Sockets

bt-mz.1024x8

export OMP_NUM_THREADS=8

my_rank=$PMI_RANK
local_rank=$(( $my_rank % $myway ))
numnode=$(( $local_rank + 1 ))

Original:
numactl -N $numnode -m $numnode $*

Modified:
if [ $local_rank -eq 0 ]; then
  numactl -N 0,3 -m 0,3 $*
else
  numactl -N 1,2 -m 1,2 $*
fi

Achieves Scalability!
• Process uses cores and memory across 2 sockets
• Suitable for 8 threads
Cray XT5

- Results obtained by the courtesy of the HPCMO Program and the Engineer Research and Development Center Major Shared Resource Center, Vicksburg, MS (http://www.erdc.hpc.mil/index)

- Cray XT5 is located at the Arctic Region Supercomputing Center (ARSC) (http://www.arsc.edu/resources/pingo)
  - 432 Cray XT5 compute nodes with
    - 32 GB of shared memory per node (4 GB per core)
    - 2 quad core 2.3 GHz AMD Opteron processors per node.
    - 1 Seastar2+ Interconnect Module per node.
  - Cray Seastar2+ Interconnect between all compute and login nodes
Cray XT5: NPB-MZ Class D Scalability

Results reported for Class D on 256-2048 cores

- SP-MZ pure MPI scales up to 1024 cores
- SP-MZ MPI/OpenMP scales to 2048 cores
- SP-MZ MPI/OpenMP outperforms pure MPI for 1024 cores
- BT-MZ MPI does not scale
- BT-MZ MPI/OpenMP scales to 2048 cores, outperforms pure MPI

Expected: #MPI processes limited

Unexpected!
Cray XT5: CrayPat Performance Analysis

- `module load xt-cr raypat`
- Compilation:
  - `ftn -fastsse -tp barcelona-64 -r8 -mp=nonuma,[trace ]`
- Instrument:
  - `pat_build -w -T TraceOmp, -g mpi,omp bt.exe bt.exe.pat`
- Execution:
  - `(export PAT_RT_HWPC {0,1,2,..})`
  - `export OMP_NUM_THREADS 4`
  - `aprun -n NPROCS -S 1 -d 4 ./bt.exe.pat`
- Generate report:
  - `pat_report -O load_balance,thread_times,program_time,mpi_callers -O profile_pe.th $1`
Cray XT5: BT-MZ Load-Balance 32x4 vs 128x1

- maximum, median, minimum PE are shown
- bt-mz.C.128x1 shows large imbalance in User and MPI time
- bt-mz.C.32x4 shows well balanced times

Table 2: Load Balance across PE's by FunctionGroup

<table>
<thead>
<tr>
<th>Time %</th>
<th>Time</th>
<th>Calls</th>
<th>Experiment=1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Group</td>
<td>PE[mmm]</td>
<td>Thread</td>
</tr>
<tr>
<td>100.0%</td>
<td>1.782603</td>
<td>18662</td>
<td>Total</td>
</tr>
<tr>
<td>86.1%</td>
<td>1.535163</td>
<td>7783</td>
<td>USER</td>
</tr>
<tr>
<td>2.7%</td>
<td>1.535987</td>
<td>6813</td>
<td>lpe.0</td>
</tr>
<tr>
<td>0.7%</td>
<td>1.535987</td>
<td>6188</td>
<td>lthread.1</td>
</tr>
<tr>
<td>0.7%</td>
<td>1.535971</td>
<td>6188</td>
<td>lthread.3</td>
</tr>
<tr>
<td>0.7%</td>
<td>1.535928</td>
<td>6188</td>
<td>lthread.2</td>
</tr>
<tr>
<td>0.7%</td>
<td>1.466954</td>
<td>6813</td>
<td>lthread.0</td>
</tr>
<tr>
<td>2.7%</td>
<td>1.535147</td>
<td>7783</td>
<td>lpe.18</td>
</tr>
<tr>
<td>0.7%</td>
<td>1.535147</td>
<td>7072</td>
<td>lthread.1</td>
</tr>
<tr>
<td>0.7%</td>
<td>1.534995</td>
<td>7072</td>
<td>lthread.3</td>
</tr>
<tr>
<td>0.7%</td>
<td>1.534968</td>
<td>7072</td>
<td>lthread.2</td>
</tr>
<tr>
<td>0.6%</td>
<td>1.290502</td>
<td>7783</td>
<td>lthread.0</td>
</tr>
<tr>
<td>2.7%</td>
<td>1.534239</td>
<td>7783</td>
<td>lpe.16</td>
</tr>
<tr>
<td>0.7%</td>
<td>1.534239</td>
<td>7072</td>
<td>lthread.1</td>
</tr>
<tr>
<td>0.7%</td>
<td>1.534101</td>
<td>7072</td>
<td>lthread.3</td>
</tr>
<tr>
<td>0.7%</td>
<td>1.534076</td>
<td>7072</td>
<td>lthread.2</td>
</tr>
<tr>
<td>0.6%</td>
<td>1.268085</td>
<td>7783</td>
<td>lthread.0</td>
</tr>
</tbody>
</table>

bt-mz-C.128x1

bt-mz-C.32x4

Hybrid Parallel Programming
Running Hybrid on Cray XT4

- Shared Memory:
  - Cache-coherent 4-way Node
- Distributed memory:
  - Network of nodes
    - Core-to-Core
    - Node-to-Node
Pitfalls:
Process and Thread Placement on Cray XT4 (1)

```
export OMP_NUM_THREADS=4
export MPICH_RANK_REORDER_DISPLAY=1
aprun -n 2 sp-mz.B.2
```

1 node, 4 cores, 8 threads

[PE_0]: rank 0 is on nid01759;
[PE_0]: rank 1 is on nid01759;

```
<table>
<thead>
<tr>
<th>SP-MZ Benchmark Completed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class  =</td>
</tr>
<tr>
<td>Size   =</td>
</tr>
<tr>
<td>Iterations =</td>
</tr>
<tr>
<td>Time in seconds =</td>
</tr>
<tr>
<td>Total processes =</td>
</tr>
<tr>
<td>Total threads =</td>
</tr>
<tr>
<td>Mop/s total =</td>
</tr>
<tr>
<td>Mop/s/thread =</td>
</tr>
<tr>
<td>Operation type =</td>
</tr>
<tr>
<td>Verification =</td>
</tr>
<tr>
<td>Version =</td>
</tr>
<tr>
<td>Compile date =</td>
</tr>
</tbody>
</table>
```

Terrible execution time because both
4-threaded MPI processes are running
on the same socket
Pitfalls:
Process and Thread Placement on Cray XT4 (2)

export OMP_NUM_THREADS=4
export MPICH_RANK_REORDER_DISPLAY=1

aprun -n 2 -N 1 sp-mz.B.2

2 nodes, 8 cores, 8 threads

[PE_0]: rank 0 is on nid01759;
[PE_0]: rank 1 is on nid01882;

---

Short execution time because both 4-way MPI processes are running on different sockets
Example Batch Script Cray XT4

Cray XT4 at ERDC:

- 1 quad-core AMD Opteron per node
- `ftn -fastsse -tp barcelona-64 -mp -o bt-mz.128`

```bash
#!/bin/csh
#PBS -q standard
#PBS -l mppwidth=512
#PBS -l walltime=00:30:00
module load xt-mpt
cd $PBS_O_WORKDIR
setenv OMP_NUM_THREADS 4
aprun -n 128 -N 1 -d 4 ./bt-mz.128

setenv OMP_NUM_THREADS 2
aprun -n 256 -N 2 -d 2 ./bt-mz.256
```

- Maximum of 4 threads per MPI process on XT4
- 4 threads per MPI process
- 2 MPI processes per node, 2 threads per MPI process
- 1 process per node allows for 4 threads per process
IBM Power 6

- Results obtained by the courtesy of the HPCMO Program and the Engineer Research and Development Center Major Shared Resource Center, Vicksburg, MS (http://www.erdc.hpc.mil/index)
- The IBM Power 6 System is located at (http://www.navo.hpc.mil/davinci_about.html)
- 150 Compute Nodes
- 32 4.7GHz Power6 Cores per Node (4800 cores total)
- 64 GBytes of dedicated memory per node
- QLOGOC Infiniband DDR interconnect
- IBM MPI: MPI 1.2 + MPI-IO
  - mpxlf_r -O4 -qarch=pwr6 -qtune=pwr6 -qsmp=omp

- Execution:
  - poe launch $PBS_O_WORKDIR./sp.C.16x4.exe

Flag was essential to achieve full compiler optimization in presence of OMP directives!
NPB-MZ Class D on IBM Power 6: Exploiting SMT for 2048 Core Results

- Results for 128-2048 cores
- Only 1024 cores were available for the experiments
- BT-MZ and SP-MZ show benefit from **Simultaneous Multithreading (SMT)**: 2048 threads on 1024 cores

```bash
#!/bin/csh
PBS -l select=32:ncpus=64:mpiprocs=NP:ompthreads=NT
```
Performance Analysis on IBM Power 6

- Compilation:
  - `mpxlf_r -O4 --qarch=pwr6 --qtune=pwr6 --qsmp=omp -pg`
- Execution:
  - `export OMP_NUM_THREADS 4`
  - `poe launch $PBS_O_WORKDIR./sp.C.16x4.exe`
  - Generates a file `gmount.MPI_RANK.out` for each MPI Process
- Generate report:
  - `gprof sp.C.16x4.exe gmon*`

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>seconds</th>
<th>self seconds</th>
<th>calls</th>
<th>ms/call</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.7</td>
<td>117.94</td>
<td>117.94</td>
<td>205245</td>
<td>0.57</td>
<td>0.57</td>
<td>.@10@x_solve@OL@1 [2]</td>
<td></td>
</tr>
<tr>
<td>14.6</td>
<td>221.14</td>
<td>103.20</td>
<td>205064</td>
<td>0.50</td>
<td>0.50</td>
<td>.@15@z_solve@OL@1 [3]</td>
<td></td>
</tr>
<tr>
<td>12.1</td>
<td>307.14</td>
<td>86.00</td>
<td>205200</td>
<td>0.42</td>
<td>0.42</td>
<td>.@12@y_solve@OL@1 [4]</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>350.83</td>
<td>43.69</td>
<td>205300</td>
<td>0.21</td>
<td>0.21</td>
<td>.@8@compute_rhs@OL@1@OL@6 [5]</td>
<td></td>
</tr>
</tbody>
</table>
Conclusions:

- **BT-MZ:**
  - Inherent workload imbalance on MPI level
  - \#nprocs = \#nzones yields poor performance
  - \#nprocs < \#zones \(\rightarrow\) room for better workload balance, but decreases parallelism
  - Hybrid MPI/OpenMP offers possibility for load-balancing while maintaining amount of parallelism
  - Best performance in hybrid mode across all platforms

- **SP-MZ:**
  - No workload imbalance on MPI level
  - Pure MPI should perform best
  - Surprising results on some platforms due to unexpected zone-assignment inherent in benchmark

- “**Best of category**” depends on many factors
  - Hard to predict
  - Good thread affinity is essential
Outline

• Introduction / Motivation
• Programming models on clusters of SMP nodes
• Case Studies / pure MPI vs hybrid MPI+OpenMP

• Practical “How-To” on hybrid programming
  
  Georg Hager, Regionales Rechenzentrum Erlangen (RRZE)

• Mismatch Problems
• Application categories that can benefit from hybrid parallelization
• Thread-safety quality of MPI libraries
• Tools for debugging and profiling MPI+OpenMP
• Summary
Hybrid Programming How-To: Overview

- A practical introduction to hybrid programming
  - How to compile and link
  - Getting a hybrid program to run on a cluster

- Running hybrid programs efficiently on multi-core clusters
  - Affinity issues
    - ccNUMA
    - Bandwidth bottlenecks
  - Intra-node MPI/OpenMP anisotropy
    - MPI communication characteristics
    - OpenMP loop startup overhead
  - Thread/process binding
How to compile, link and run

• Use appropriate OpenMP compiler switch (-openmp, -xopenmp, -mp, -qsmp=openmp, …) and MPI compiler script (if available)
• Link with MPI library
  – Usually wrapped in MPI compiler script
  – If required, specify to link against thread-safe MPI library
    • Often automatic when OpenMP or auto-parallelization is switched on
• Running the code
  – Highly non-portable! Consult system docs! (if available…)
  – If you are on your own, consider the following points
  – Make sure OMP_NUM_THREADS etc. is available on all MPI processes
    • Start “env VAR=VALUE ... <YOUR BINARY>” instead of your binary alone
    • Use Pete Wyckoff’s mpiexec MPI launcher (see below):
      http://www.osc.edu/~pw/mpiexec
  – Figure out how to start less MPI processes than cores on your nodes
Some examples for compilation and execution (1)

- **NEC SX9**
  - NEC SX9 compiler
  - `mpif90 -C hopt -P openmp ... # -ftrace for profiling info`
  - Execution:

    ```
    $ export OMP_NUM_THREADS=<num_threads>
    $ MPIEXPORT="OMP_NUM_THREADS"
    $ mpirun -nn <# MPI procs per node> -nnp <# of nodes> a.out
    ```

- **Standard Intel Xeon cluster (e.g. @HLRS):**
  - Intel Compiler
  - `mpif90 -openmp ...`
  - Execution (handling of `OMP_NUM_THREADS`, see next slide):

    ```
    $ mpirun_ssh -np <num MPI procs> -hostfile machines a.out
    ```
Handling of OMP_NUM_THREADS

- **without** any support by mpirun:
  - E.g. with mpich-1
  - Problem:
    mpirun has no features to export environment variables to the via ssh automatically started MPI processes
  - Solution: Set
    `export OMP_NUM_THREADS=<# threads per MPI process>`
    in ~/.bashrc (if a bash is used as login shell)
  - If you want to set OMP_NUM_THREADS individually when starting the MPI processes:
    - Add
      ```bash
test -s ~/myexports && . ~/myexports
      in your ~/.bashrc
      ```
    - Add
      ```bash
echo '${OMP_NUM_THREADS=<# threads per MPI process>}' > ~/myexports
      ```
      before invoking mpirun
    - Caution: Several invocations of mpirun cannot be executed at the same time with this trick!
Handling of OMP_NUM_THREADS (continued)

- **with** support by OpenMPI `-x` option:
  
  ```
  export OMP_NUM_THREADS= <# threads per MPI process>
  mpiexec -x OMP_NUM_THREADS -n <# MPI processes> ./executable
  ```

Some examples for compilation and execution (3)
Some examples for compilation and execution (4)

- **Sun Constellation Cluster:**
  - `mpif90 -fastsse -tp barcelona-64 -mp ...`
  - SGE Batch System
  - `setenv OMP_NUM_THREADS`
  - `ibrun numactl.sh a.out`
  - Details see TACC Ranger User Guide (www.tacc.utexas.edu/services/userguides/ranger/#numactl)

- **Cray XT5:**
  - `ftn -fastsse -tp barcelona-64 -mp=nonuma ...`
  - `aprun -n nprocs -N nprocs_per_node a.out`
Interlude: Advantages of mpiexec

- Uses PBS/Torque Task Manager (“TM”) interface to spawn MPI processes on nodes
  - As opposed to starting remote processes with ssh/rsh:
    - Correct CPU time accounting in batch system
    - Faster startup
    - Safe process termination
    - Understands PBS per-job nodefile
    - Allowing password-less user login not required between nodes
  - Support for many different types of MPI
    - All MPICHs, MVAPICHs, Intel MPI, …
  - Interfaces directly with batch system to determine number of procs
  - Downside: If you don’t use PBS or Torque, you’re out of luck…

- Provisions for starting less processes per node than available cores
  - Required for hybrid programming
  - “-pernode” and “-npernode #” options – does not require messing around with nodefiles
Running the code

- Example for using mpiexec on a dual-socket dual-core cluster:

  $ export OMP_NUM_THREADS=4
  $ mpiexec -pernode ./a.out

- Same but 2 MPI processes per node:

  $ export OMP_NUM_THREADS=2
  $ mpiexec -npernode 2 ./a.out

- Pure MPI:

  $ export OMP_NUM_THREADS=1 # or nothing if serial code
  $ mpiexec ./a.out
Running the code *efficiently*?

- Symmetric, UMA-type compute nodes have become rare animals
  - NEC SX
  - Intel 1-socket ("Port Townsend/Melstone") – see case studies
  - Hitachi SR8000, IBM SP2, single-core multi-socket Intel Xeon… (all dead)

- Instead, systems have become “non-isotropic” on the node level
  - ccNUMA (AMD Opteron, SGI Altix, IBM Power6 (p575), larger Sun Enterprise systems, Intel Nehalem)
  - Multi-core, multi-socket
    - Shared vs. separate caches
    - Multi-chip vs. single-chip
    - Separate/shared buses
Issues for running code efficiently on “non-isotropic” nodes

- ccNUMA locality effects
  - Penalties for inter-LD access
  - Impact of contention
  - Consequences of file I/O for page placement
  - Placement of MPI buffers

- Multi-core / multi-socket anisotropy effects
  - Bandwidth bottlenecks, shared caches
  - Intra-node MPI performance
    - Core ↔ core vs. socket ↔ socket
  - OpenMP loop overhead depends on mutual position of threads in team
A short introduction to ccNUMA

- ccNUMA:
  - whole memory is **transparently accessible** by all processors
  - but **physically distributed**
  - with **varying bandwidth and latency**
  - and **potential contention** (shared memory paths)
Example: HP DL585 G5
4-socket ccNUMA Opteron 8220 Server

- **CPU**
  - 64 kB L1 per core
  - 1 MB L2 per core
  - No shared caches
  - On-chip memory controller (MI)
  - 10.6 GB/s local memory bandwidth

- **HyperTransport 1000 network**
  - 4 GB/s per link per direction

- **3 distance categories** for core-to-memory connections:
  - same LD
  - 1 hop
  - 2 hops

- **Q1**: What are the real penalties for non-local accesses?
- **Q2**: What is the impact of contention?
Effect of non-local access on HP DL585 G5:
Serial vector triad \( A(:) = B(:) + C(:) \times D(:) \)
Contestation vs. parallel access on HP DL585 G5:
OpenMP vector triad $A(:)=B(:)+C(:)*D(:)$

In-cache performance unharmed by ccNUMA

Affinity matters!

Single LD saturated by 2 cores!

Perfect scaling across LDs
ccNUMA Memory Locality Problems

- Locality of reference is key to scalable performance on ccNUMA
  - Less of a problem with pure MPI, but see below
- What factors can destroy locality?
- MPI programming:
  - processes lose their association with the CPU the mapping took place on originally
  - OS kernel tries to maintain strong affinity, but sometimes fails
- Shared Memory Programming (OpenMP, hybrid):
  - threads losing association with the CPU the mapping took place on originally
  - improper initialization of distributed data
  - Lots of extra threads are running on a node, especially for hybrid
- All cases:
  - Other agents (e.g., OS kernel) may fill memory with data that prevents optimal placement of user data
Avoiding locality problems

• How can we make sure that memory ends up where it is close to the CPU that uses it?
  – See the following slides

• How can we make sure that it stays that way throughout program execution?
  – See end of section
Solving Memory Locality Problems: First Touch

- "Golden Rule" of ccNUMA:
  A memory page gets mapped into the local memory of the processor that first touches it!
  - Except if there is not enough local memory available
  - this might be a problem, see later
  - Some OSs allow to influence placement in more direct ways
    - cf. libnuma (Linux), MPO (Solaris), ...
- Caveat: "touch" means "write", not "allocate"
- Example:

  ```c
  double *huge = (double*)malloc(N*sizeof(double));
  // memory not mapped yet
  for(i=0; i<N; i++) // or i+=PAGE_SIZE
    huge[i] = 0.0; // mapping takes place here!
  ```

- It is sufficient to touch a single item to map the entire page
ccNUMA problems beyond first touch

- OS uses part of main memory for disk buffer (FS) cache
  - If FS cache fills part of memory, apps will probably allocate from foreign domains
  - \(\rightarrow\) non-local access!
  - Locality problem even on hybrid and pure MPI with “asymmetric” file I/O, i.e. if not all MPI processes perform I/O

- Remedies
  - Drop FS cache pages after user job has run (admin’s job)
    - Only prevents cross-job buffer cache “heritage”
  - “Sweeper” code (run by user)
  - Flush buffer cache after I/O if necessary (“sync” is not sufficient!)
ccNUMA problems beyond first touch

- Real-world example: ccNUMA vs. UMA and the Linux buffer cache
- Compare two 4-way systems: AMD Opteron ccNUMA vs. Intel UMA, 4 GB main memory
- Run 4 concurrent triads (512 MB each) after writing a large file
- Report performance vs. file size
- Drop FS cache after each data point
Intra-node MPI characteristics: IMB Ping-Pong benchmark

- Code (to be run on 2 processors):

  wc = MPI_WTIME()

do i=1,NREPEAT

  if(rank.eq.0) then
    MPI_SEND(buffer,N,MPI_BYTE,1,0,MPI_COMM_WORLD,ierr)
    MPI_RECV(buffer,N,MPI_BYTE,1,0,MPI_COMM_WORLD, &
    status,ierr)
  else
    MPI_RECV(...)
    MPI_SEND(...)
  endif
endo

wc = MPI_WTIME() - wc

- Intranode (1S): mpirun -np 2 -pin "1 3" ./a.out
- Intranode (2S): mpirun -np 2 -pin "2 3" ./a.out
- Internode: mpirun -np 2 -pernode ./a.out
IMB Ping-Pong: Latency

*Intra-node vs. Inter-node on Woodcrest DDR-IB cluster* (Intel MPI 3.1)

Affinity matters!
IMB Ping-Pong: Bandwidth Characteristics

Intra-node vs. Inter-node on Woodcrest DDR-IB cluster (Intel MPI 3.1)

Affinity matters!
OpenMP Overhead

- As with intra-node MPI, OpenMP loop start overhead varies with the mutual position of threads in a team
- Possible variations
  - Intra-socket vs. inter-socket
  - Different overhead for "parallel for" vs. plain "for"
  - If one multi-threaded MPI process spans multiple sockets,
    - ... are neighboring threads on neighboring cores?
    - ... or are threads distributed "round-robin" across cores?
- Test benchmark: **Vector triad**

```c
#pragma omp parallel
for(int j=0; j < NITER; j++){
    #pragma omp (parallel) for
    for(i=0; i < N; ++i)
        a[i]=b[i]+c[i]*d[i];
    if(OBSCURE)
        dummy(a,b,c,d);
}
```

Look at performance for small array sizes!
OpenMP Overhead

Hybrid Parallel Programming
Slide 67 / 151
Rabenseifner, Hager, Jost

Nomenclature:

1S/2S
1-/2-socket

RR
round-robin

SS
socket-socket

inner
parallel on inner loop

OMP overhead is comparable to MPI latency!

Affinity matters!
Thread/Process Affinity ("Pinning")

- Highly OS-dependent system calls
  - But available on all systems
    - Linux: `sched_setaffinity()`, PLPA (see below)
    - Solaris: `processor_bind()`
    - Windows: `SetThreadAffinityMask()`
  - ... 
- Support for "semi-automatic" pinning in some compilers/environments
  - Intel compilers > V9.1 (`KMP_AFFINITY` environment variable)
  - Pathscale
  - SGI Altix `dplace` (works with logical CPU numbers!)
  - Generic Linux: `taskset`, `numactl`
- Affinity awareness in MPI libraries
  - SGI MPT
  - OpenMPI
  - Intel MPI
  - ... 

Widely usable example: Using PLPA under Linux!
Process/Thread Binding With PLPA on Linux:
http://www.open-mpi.org/software/plpa/

- **Portable Linux Processor Affinity**
- Wrapper library for `sched_*affinity()` functions
  - Robust against changes in kernel API
- Example for **pure OpenMP**: Pinning of threads

```
#include <plpa.h>
...
#pragma omp parallel
{
  #pragma omp critical
  {
    if(PLPA_NAME(api_probe)() != PLPA_PROBE_OK) {
      cerr << "PLPA failed!" << endl; exit(1);
    }
    plpa_cpu_set_t msk;
    PLPA_CPU_ZERO(&msk);
    int cpu = omp_get_thread_num();
    PLPA_CPU_SET(cpu, &msk);
    PLPA_NAME(sched_setaffinity)((pid_t)0, sizeof(cpu_set_t), &msk);
  }
```

Care about correct core numbering! 0…N-1 is not always contiguous! If required, reorder by a map:
```
cpu = map[cpu];
```

Pinning available?

Which CPU to run on?

Pin “me”
Process/Thread Binding With PLPA

- Example for pure MPI: Process pinning
  - Bind MPI processes to cores in a cluster of 2x2-core machines

  ```
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  int mask = (rank % 4);
  PLPA_CPU_SET(mask,&msk);
  PLPA_NAME(sched_setaffinity)((pid_t)0,
  sizeof(cpu_set_t), &msk);
  ```

- Hybrid case:

  ```
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  #pragma omp parallel
  {
    plpa_cpu_set_t msk;
    PLPA_CPU_ZERO(&msk);
    int cpu = (rank % MPI_PROCESSES_PER_NODE)*omp_num_threads
    + omp_get_thread_num();
    PLPA_CPU_SET(cpu,&msk);
    PLPA_NAME(sched_setaffinity)((pid_t)0, sizeof(cpu_set_t), &msk);
  }
  ```
Example: 3D Jacobi Solver

*Basic implementation (2 arrays; no blocking etc…)*

\[
\begin{align*}
\text{do } k &= 1, N_k \\
\text{do } j &= 1, N_j \\
\text{do } i &= 1, N_i \\
\quad y(i,j,k) &= a \cdot x(i,j,k) + b \cdot \\
& \quad (x(i-1,j,k) + x(i+1,j,k) + x(i,j-1,k) \\
& \quad + x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1))
\end{align*}
\]

Performance metric:
Million Lattice Site Updates per second (MLUPs)

Equivalent MFLOPs:
8 FLOP/LUP * MLUPs

MPI Parallelization by
- Domain Decomposition
- Halo cells
- Data Exchange through cyclic SendReceive operation
MPI/OpenMP Parallelization – 3D Jacobi

- Cubic 3D computational domain with periodic BCs in all directions
- Use single-node IB/GE cluster with one dual-core chip per node
- Homogeneous distribution of workload, e.g. on 8 procs
Performance Data for 3D MPI/hybrid Jacobi

Strong scaling, $N^3 = 480^3$

**FullHybrid**: Thread 0: Communication + boundary cell updates
Thread 1: Inner cell updates

**Performance model**

$$T = T_{COMP} + T_{COMM}$$

$$T_{COMP} = \frac{N^3}{P_0}$$

$$T_{COMM} = \frac{V_{data}}{BW}$$

$$P_0 = 150 \text{ MLUP/s}$$

$$BW(GE) = 100 \text{ MByte/s}$$

$$V_{data} = \text{Data volume of halo exchange}$$

**Performance estimate (GE) for } n \text{ nodes:**

$$P(n) = \frac{N^3}{((T_{COMP}/n) + T_{COMM}(n))}$$
Example: Sparse MVM

*JDS parallel sparse matrix-vector multiply – storage scheme*

- **val[]** stores all the nonzeros (length \( N_{nz} \))
- **col_idx[]** stores the column index of each nonzero (length \( N_{nz} \))
- **jd_ptr[]** stores the starting index of each new jagged diagonal in **val[]**
- **perm[]** holds the permutation map (length \( N_r \))

(JDS = Jagged Diagonal Storage)
JDS Sparse MVM – Kernel Code

OpenMP parallelization

- Implement \( c(:) = m(:,:) \ast b(:) \)
- Operation count = \( 2N_{nz} \)

\[
\begin{align*}
\text{do diag}=1, \text{zmax} \\
\quad \text{diagLen} &= \text{jd_ptr(diag+1)} - \text{jd_ptr(diag)} \\
\quad \text{offset} &= \text{jd_ptr(diag)} - 1 \\
\text{!!$OMP PARALLEL DO} \\
\quad \text{do } i=1, \text{diagLen} \\
\quad \quad c(i) &= c(i) + \text{val(offset+i)} \ast b(\text{col_idx(offset+i)}) \\
\text{enddo} \\
\text{!!$OMP END PARALLEL DO} \\
\text{enddo}
\end{align*}
\]

- Long inner loop (max. \( N_r \)): OpenMP parallelization / vectorization
- Short outer loop (number of jagged diagonals)
- Multiple accesses to each element of result vector \( c[\] \)
  - optimization potential!
- Stride-1 access to matrix data in \( \text{val[} \]
- Indexed (indirect) access to RHS vector \( b[\] \)
JDS Sparse MVM

MPI parallelization

Avoid mixing of local and non-local diagonals:

1. Shift within local subblock
2. Fill local subblock with non-local elements from the right
JDS Sparse MVM

Parallel MVM implementations: MPP

- One MPI process per processor
- Non-blocking MPI communication
- Potential overlap of communication and computation
  - However, MPI progress is only possible inside MPI calls on many implementations
- SMP Clusters: Intra-node and inter-node MPI

1. Start: isend/irecv
2. Release local diags
3. Compute MVM with diags released
4. Test: irecv
5. Release diags ?
6. irecv ?
JDS Sparse MVM

Parallel MVM implementations: Hybrid

VECTOR mode:
- Automatic parallel. of inner i loop (data parallel)
- Single threaded MPI calls

TASK mode:
- Functional parallelism: Simulate asynchronous data transfer! (OpenMP)
- Release list - LOCK
- Single threaded MPI calls
- Optional: Comm. Thread executes configurable fraction of work (load = 0...1)
JDS Sparse MVM: 
Performance and scalability on two different platforms

Opteron 270 2 GHz

Xeon 5160 3 GHz

no NUMA placement!

71 · 10^6 nonzeroes

Hybrid Parallel Program
Outline

• Introduction / Motivation
• Programming models on clusters of SMP nodes
• Case Studies / pure MPI vs hybrid MPI+OpenMP
• Practical “How-To” on hybrid programming

• Mismatch Problems

• Opportunities:
  Application categories that can benefit from hybrid parallelization
• Thread-safety quality of MPI libraries
• Tools for debugging and profiling MPI+OpenMP
• Summary
Mismatch Problems

- None of the programming models fits to the hierarchical hardware (cluster of SMP nodes)
- Several mismatch problems → following slides
- Benefit through hybrid programming → Opportunities, see next section
- Quantitative implications → depends on your application

<table>
<thead>
<tr>
<th>Examples:</th>
<th>No.1</th>
<th>No.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benefit through hybrid</td>
<td>30%</td>
<td>10%</td>
</tr>
<tr>
<td>(see next section)</td>
<td>-10%</td>
<td>-25%</td>
</tr>
<tr>
<td>Loss by mismatch problems</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>+20%</td>
<td>-15%</td>
</tr>
</tbody>
</table>

In most cases: Both categories!
The Topology Problem with

Application example on 80 cores:
- Cartesian application with $5 \times 16 = 80$ sub-domains
- On system with $10 \times$ dual socket $\times$ quad-core

Sequential ranking of MPI_COMM_WORLD

Does it matter?
The Topology Problem with pure MPI
one MPI process
on each core

Application example on 80 cores:
• Cartesian application with 5 x 16 = 80 sub-domains
• On system with 10 x dual socket x quad-core

32 x inter-node connections per node
0 x inter-socket connection per node

Round robin ranking of MPI_COMM_WORLD

Never trust the default !!!
The Topology Problem with

Application example on 80 cores:
- Cartesian application with $5 \times 16 = 80$ sub-domains
- On system with $10 \times$ dual socket $\times$ quad-core

- 10 x inter-node connections per node
- 4 x inter-socket connection per node

**Bad** affinity of cores to thread ranks

Two levels of domain decomposition
The Topology Problem with pure MPI

Application example on 80 cores:
- Cartesian application with $5 \times 16 = 80$ sub-domains
- On system with $10 \times$ dual socket $\times$ quad-core

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
</tr>
<tr>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
</tr>
<tr>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
</tr>
<tr>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
</tr>
</tbody>
</table>

- 10 x inter-node connections per node
- 2 x inter-socket connection per node

Good affinity of cores to thread ranks

Two levels of domain decomposition
The Topology Problem with

Problem
- Does application topology inside of SMP parallelization fit on inner hardware topology of each SMP node?

Solutions:
- Domain decomposition inside of each thread-parallel MPI process, and
- first touch strategy with OpenMP

Successful examples:
- Multi-Zone NAS Parallel Benchmarks (MZ-NPB)
The Topology Problem with hybrid MPI+OpenMP

Application example:
- Same Cartesian application aspect ratio: 5 x 16
- On system with 10 x dual socket x quad-core
- 2 x 5 domain decomposition

MPI: inter-node communication
OpenMP: inside of each SMP node

- 3 x inter-node connections per node, but ~ 4 x more traffic
- 2 x inter-socket connection per node

Affinity of cores to thread ranks !!!
The Mapping Problem with **mixed model**

**Problem**
- Where are your processes and threads really located?

**Solutions:**
- Depends on your platform,
- e.g., `lbrun` `numactl` option on Sun

As seen in case-study on Sun Constellation Cluster Ranger with BT-MZ and SP-MZ
Unnecessary intra-node communication

Problem:
- If several MPI process on each SMP node
  → unnecessary intra-node communication

Solution:
- Only one MPI process per SMP node

Remarks:
- MPI library must use appropriate fabrics / protocol for intra-node communication
- Intra-node bandwidth higher than inter-node bandwidth
  → problem may be small
- MPI implementation may cause unnecessary data copying
  → waste of memory bandwidth

Quality aspects of the MPI library

Mixed model
(several multi-threaded MPI processes per SMP node)
Sleeping threads and network saturation with Masteronly

MPI only outside of parallel regions

Problem 1:
- Can the master thread saturate the network?
Solution:
- If not, use mixed model
- i.e., several MPI processes per SMP node

Problem 2:
- Sleeping threads are wasting CPU time
Solution:
- Overlapping of computation and communication

Problem 1&2 together:
- Producing more idle time through lousy bandwidth of master thread

for (iteration ....)
{
    #pragma omp parallel
    numerical code
    /*end omp parallel */

    /* on master thread only */
    MPI_Send (original data to halo areas in other SMP nodes)
    MPI_Recv (halo data from the neighbors)
} /*end for loop

Node Interconnect
OpenMP: Additional Overhead & Pitfalls

- Using OpenMP
  - may prohibit compiler optimization
  - may cause significant loss of computational performance
- Thread fork / join overhead
- On ccNUMA SMP nodes:
  - Loss of performance due to missing memory page locality or missing first touch strategy
  - E.g. with the masteronly scheme:
    - One thread produces data
    - Master thread sends the data with MPI
    - data may be internally communicated from one memory to the other one
- Amdahl’s law for each level of parallelism
- Using MPI-parallel application libraries? → Are they prepared for hybrid?

See, e.g., the necessary -O4 flag with mpxlf_r on IBM Power6 systems
Overlapping Communication and Computation

MPI communication by one or a few threads while other threads are computing

Three problems:

• the application problem:
  – one must separate application into:
    • code that can run before the halo data is received
    • code that needs halo data
  ➔ very hard to do !!!

• the thread-rank problem:
  – comm. / comp. via thread-rank
  – cannot use work-sharing directives
  ➔ loss of major OpenMP support (see next slide)

• the load balancing problem

```c
if (my_thread_rank < 1) {
    MPI_Send/Recv....
} else {
    my_range = (high-low-1) / (num_threads-1) + 1;
    my_low = low + (my_thread_rank+1)*my_range;
    my_high = high+ (my_thread_rank+1+1)*my_range;
    my_high = max(high, my_high)
    for (i=my_low; i<my_high; i++) {
        ....
    }
} 
```
Overlapping Communication and Computation

MPI communication by one or a few threads while other threads are computing

**Subteams**

- Important proposal for OpenMP 3.x or OpenMP 4.x


```c
#pragma omp parallel
{
#pragma omp single onthreads( 0 )
{
    MPI_Send/Recv....
}
#pragma omp for onthreads( 1 : omp_get_numthreads()-1 )
    for (.........)
        {
            /* work without halo information */
        } /* barrier at the end is only inside of the subteam */
... 
#pragma omp barrier
#pragma omp for
    for (.........)
        {
            /* work based on halo information */
        }
} /*end omp parallel */
```
Parallel Programming Models on Hybrid Platforms

- **pure MPI**
  - one MPI process on each core

- **hybrid MPI+OpenMP**
  - MPI: inter-node communication
  - OpenMP: inside of each SMP node

- **OpenMP only**
  - distributed virtual shared memory

### Communication and Computation

- **No overlap of Comm. + Comp.**
  - MPI only outside of parallel regions of the numerical application code

- **Overlapping Comm. + Comp.**
  - MPI communication by one or a few threads while other threads are computing

### MPI Deployment

- **Masteronly**
  - MPI only outside of parallel regions

- **Multiple/only**
  - appl. threads
  - inside of MPI

- **Funnelleled**
  - MPI only on master-thread

- **Multiple**
  - more than one thread may communicate

### Load Balancing Strategies

- **Different strategies to simplify the load balancing**

- **Funnelleled & Reserved**
  - reserved thread for communication

- **Funnelleled with Full Load Balancing**

- **Multiple & Reserved**
  - reserved threads for communication

- **Multiple with Full Load Balancing**
Experiment: Matrix-vector-multiply (MVM)

- Jacobi-Davidson-Solver on IBM SP Power3 nodes with 16 CPUs per node
- funneled & reserved is always faster in this experiments
- Reason:
  Memory bandwidth is already saturated by 15 CPUs, see inset
- Inset:
  Speedup on 1 SMP node using different number of threads

Source: R. Rabenseifner, G. Wellein:
Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures.
OpenMP/DSM

- Distributed shared memory (DSM)
- Distributed virtual shared memory (DVSM)
- Shared virtual memory (SVM)

Principles
- emulates a shared memory
- on distributed memory hardware

Implementations
- e.g., Intel® Cluster OpenMP
Basic idea:

- Between OpenMP barriers, data exchange is not necessary, i.e., visibility of data modifications to other threads only after synchronization.
- When a page of sharable memory is not up-to-date, it becomes **protected**.
- Any access then faults (SIGSEGV) into Cluster OpenMP runtime library, which requests info from remote nodes and updates the page.
- Protection is removed from page.
- Instruction causing the fault is re-started, this time successfully accessing the data.
Comparison:
MPI based parallelization ↔ DSM

- MPI based:
  - Potential of boundary exchange between two domains in one large message
    → Dominated by **bandwidth** of the network

- DSM based (e.g. Intel® Cluster OpenMP):
  - Additional latency based overhead in each barrier
    → May be marginal
  - Communication of **updated data of pages**
    → Not all of this data may be needed
    → i.e., too much data is transferred
    → Packages may be too small
    → Significant latency
  - Communication not oriented on boundaries of a domain decomposition
    → Probably more data must be transferred than necessary

**by rule of thumb:**
Communication may be 10 times slower than with MPI
Comparing results with heat example

- Normal OpenMP on shared memory (ccNUMA) NEC TX-7

[Graph showing speedup for different grid sizes with OpenMP on NEC TX-7]
Heat example: Cluster OpenMP Efficiency

- Cluster OpenMP on a Dual-Xeon cluster

Efficiency only with small communication foot-print

Up to 3 CPUs with 3000x3000

No speedup with 1000x1000

Second CPU only usable in small cases

Terrible with non-default schedule
Back to the mixed model – an Example

- Topology-problem solved: Only horizontal inter-node comm.
- Still intra-node communication
- Several threads per SMP node are communicating in parallel: → network saturation is possible
- Additional OpenMP overhead
- With Master only style: 75% of the threads sleep while master thread communicates
- With Overlapping Comm. & Comp.: Master thread should be reserved for communication only partially – otherwise too expensive
- MPI library must support
  - Multiple threads
  - Two fabrics (shmep + internode)
No silver bullet

• The analyzed programming models do not fit on hybrid architectures
  – whether drawbacks are minor or major
    → depends on applications’ needs
  – But there are major opportunities → next section

• In the NPB-MZ case-studies
  – We tried to use optimal parallel environment
    • for pure MPI
    • for hybrid MPI+OpenMP
  – i.e., the developers of the MZ codes and we tried to minimize the mismatch problems
  → the opportunities in next section dominated the comparisons
Outline

• Introduction / Motivation
• Programming models on clusters of SMP nodes
• Case Studies / pure MPI vs hybrid MPI+OpenMP
• Practical “How-To” on hybrid programming
• Mismatch Problems

• Opportunities:
  Application categories that can benefit from hybrid parallelization

• Thread-safety quality of MPI libraries
• Tools for debugging and profiling MPI+OpenMP
• Summary
Nested Parallelism

- Example NPB: BT-MZ  (Block tridiagonal simulated CFD application)
  - Outer loop:
    - limited number of zones  $\rightarrow$ limited parallelism
    - zones with different workload  $\rightarrow$ speedup $< \frac{\text{Sum of workload of all zones}}{\text{Max workload of a zone}}$
  - Inner loop:
    - OpenMP parallelized (static schedule)
    - Not suitable for distributed memory parallelization

- Principles:
  - Limited parallelism on outer level
  - Additional inner level of parallelism
  - Inner level not suitable for MPI
  - Inner level may be suitable for static OpenMP worksharing
Load-Balancing
(on same or different level of parallelism)

- OpenMP enables
  - Cheap *dynamic* and *guided* load-balancing
  - Just a parallelization option (clause on omp for / do directive)
  - Without additional software effort
  - Without explicit data movement

- On MPI level
  - *Dynamic load balancing* requires
    moving of parts of the data structure through the network
  - Significant runtime overhead
  - Complicated software / therefore not implemented

- **MPI & OpenMP**
  - Simple static load-balancing on MPI level,
    dynamic or guided on OpenMP level
    \[ \text{medium quality} \]
    \[ \text{cheap implementation} \]
Memory consumption

• Shared nothing
  – Heroic theory
  – In practice: Some data is duplicated

• **MPI & OpenMP**
  With n threads per MPI process:
  – Duplicated data may be reduced by factor n
Memory consumption  (continued)

- Future:
  With 100+ cores per chip the memory per core is limited.
  - Data reduction through usage of shared memory may be a key issue
  - Domain decomposition on each hardware level
    - **Maximizes**
      - Data locality
      - Cache reuse
    - **Minimizes**
      - ccNUMA accesses
      - Message passing
    - No halos between domains inside of SMP node
      - **Minimizes**
        - Memory consumption
How many multi-threaded MPI processes per SMP node

- SMP node = with \textbf{m sockets} and \textbf{n cores/socket}
- How many threads (i.e., cores) per MPI process?
  - Too many threads per MPI process
    - overlapping of MPI and computation may be necessary,
      - some NICs unused?
  - Too few threads
    - too much memory consumption (see previous slides)
- Optimum
  - somewhere between 1 and m x n
Opportunities, if MPI speedup is limited due to algorithmic problems

- Algorithmic opportunities due to larger physical domains inside of each MPI process
  - If multigrid algorithm only inside of MPI processes
  - If separate preconditioning inside of MPI nodes and between MPI nodes
  - If MPI domain decomposition is based on physical zones
To overcome MPI scaling problems

- Reduced number of MPI messages, reduced aggregated message size compared to pure MPI
- MPI has a few scaling problems
  - Handling of more than 10,000 processes
  - Irregular Collectives: MPI_....v(), e.g. MPI_Gatherv()
    - Scaling applications should not use MPI_....v() routines
  - MPI-2.1 Graph topology (MPI_Graph_create)
    - MPI-2.2 MPI_Dist_graph_create_adjacent
  - Creation of sub-communicators with MPI_Comm_create
    - MPI-2.2 introduces a new scaling meaning of MPI_Comm_create
  - ... SC09-BOF, Wednesday 05:30PM - 07:00PM, Room D135-136
    - MPI Forum: Preview of the MPI 3 Standard (Comment Session)
- Hybrid programming reduces all these problems (due to a smaller number of processes)
Summary: Opportunities of hybrid parallelization (MPI & OpenMP)

• Nested Parallelism
  → Outer loop with MPI / inner loop with OpenMP

• Load-Balancing
  → Using OpenMP *dynamic* and *guided* worksharing

• Memory consumption
  → Significantly reduction of replicated data on MPI level

• Opportunities, if MPI speedup is limited due to algorithmic problem
  → Significantly reduced number of MPI processes

• Reduced MPI scaling problems
  → Significantly reduced number of MPI processes
Outline

• Introduction / Motivation
• Programming models on clusters of SMP nodes
• Case Studies / pure MPI vs hybrid MPI+OpenMP
• Practical “How-To” on hybrid programming
• Mismatch Problems
• Opportunities:
  Application categories that can benefit from hybrid parallelization
  
• Thread-safety quality of MPI libraries

• Tools for debugging and profiling MPI+OpenMP
• Summary
Thread-safety of MPI Libraries

- Make most powerful usage of hierarchical structure of hardware:
- Efficient programming of clusters of SMP nodes
  
  **SMP nodes:**
  - Dual/multi core CPUs
  - Multi CPU shared memory
  - Multi CPU ccNUMA
  - Any mixture with shared memory programming model

- No restriction to the usage of OpenMP for intranode-parallelism:
  - OpenMP does not (yet) offer binding threads to processors
  - OpenMP does not guarantee thread-ids to stay fixed.

- OpenMP is based on the implementation dependant thread-library: LinuxThreads, NPTL, SolarisThreads.

Courtesy of Rainer Keller, HLRS and ORNL
MPI rules with OpenMP / Automatic SMP-parallelization

• Special MPI-2 Init for multi-threaded MPI processes:

```c
int MPI_Init_thread( int * argc, char ** argv[],
                    int thread_level_required,
                    int * thead_level_provided);
int MPI_Query_thread( int * thread_level_provided);
int MPI_Is_main_thread(int * flag);
```

• REQUIRED values (increasing order):
  – **MPI_THREAD_SINGLE**: Only one thread will execute
  – **THREAD_MASTERONLY**:
    (virtual value,
     not part of the standard)
    MPI processes may be multi-threaded,
    but only master thread will make MPI-calls
    AND only while other threads are sleeping
  – **MPI_THREAD_FUNNELED**: Only master thread will make MPI-calls
  – **MPI_THREAD_SERIALIZED**: Multiple threads may make MPI-calls,
    but only one at a time
  – **MPI_THREAD_MULTIPLE**: Multiple threads may call MPI,
    with no restrictions

• returned *provided* may be less than REQUIRED by the application
Calling MPI inside of OMP MASTER

- Inside of a parallel region, with "OMP MASTER"
- Requires MPI_THREAD_FUNNELED, i.e., only master thread will make MPI-calls
- **Caution:** There isn’t any synchronization with “OMP MASTER”! Therefore, “OMP BARRIER” normally necessary to guarantee, that data or buffer space from/for other threads is available before/after the MPI call!

```
!$OMP BARRIER  #pragma omp barrier
!$OMP MASTER   #pragma omp master
   call MPI_Xxx(...)  MPI_Xxx(...);
!$OMP END MASTER
!$OMP BARRIER  #pragma omp barrier
```

- But this implies that all other threads are sleeping!
- The additional barrier implies also the necessary cache flush!
... the barrier is necessary –
example with MPI_Recv

```c
#pragma omp parallel
{
    #pragma omp for nowait
    for (i=0; i<1000; i++)
        a[i] = buf[i];

    #pragma omp barrier
    #pragma omp master
    MPI_Recv(buf,...);
    #pragma omp barrier

    #pragma omp for nowait
    for (i=0; i<1000; i++)
        c[i] = buf[i];
}
/* omp end parallel */
```
Thread support in MPI libraries

- The following MPI libraries offer thread support:

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Thread support level</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIch-1.2.7p1</td>
<td>Always announces MPI_THREAD_FUNNELED.</td>
</tr>
<tr>
<td>MPIch2-1.0.8</td>
<td>ch3:sock supports MPI_THREAD_MULTIPLE</td>
</tr>
<tr>
<td></td>
<td>ch:nemesis has “Initial Thread-support”</td>
</tr>
<tr>
<td></td>
<td>ch3:nemesis (default) has MPI_THREAD_MULTIPLE</td>
</tr>
<tr>
<td>MPIch2-1.1.0a2</td>
<td>ch3:nemesis (default) has MPI_THREAD_MULTIPLE</td>
</tr>
<tr>
<td>Intel MPI 3.1</td>
<td>Full MPI_THREAD_MULTIPLE</td>
</tr>
<tr>
<td>SciCortex MPI</td>
<td>MPI_THREAD_FUNNELED</td>
</tr>
<tr>
<td>HP MPI-2.2.7</td>
<td>Full MPI_THREAD_MULTIPLE (with libmtmpi)</td>
</tr>
<tr>
<td>SGI MPT-1.14</td>
<td>Not thread-safe?</td>
</tr>
<tr>
<td>IBM MPI</td>
<td>Full MPI_THREAD_MULTIPLE</td>
</tr>
<tr>
<td>Nec MPI/SX</td>
<td>MPI_THREAD_SERIALIZED</td>
</tr>
</tbody>
</table>

- Testsuites for thread-safety may still discover bugs in the MPI libraries

Courtesy of Rainer Keller, HLRS and ORNL
Thread support within Open MPI

- In order to enable thread support in Open MPI, configure with:
  
  ```
  configure --enable-mpi-threads
  ```

- This turns on:
  - Support for full `MPI_THREAD_MULTIPLE`
  - Internal checks when run with threads (`--enable-debug`)

  ```
  configure --enable-mpi-threads --enable-progress-threads
  ```

- This (additionally) turns on:
  - Progress threads to asynchronously transfer/receive data per network BTL.

- Additional Feature:
  - Compiling with debugging support, but without threads will check for recursive locking
Outline

- Introduction / Motivation
- Programming models on clusters of SMP nodes
- Case Studies / pure MPI vs hybrid MPI+OpenMP
- Practical “How-To” on hybrid programming
- Mismatch Problems
- Opportunities:
  Application categories that can benefit from hybrid parallelization
- Thread-safety quality of MPI libraries

- **Tools for debugging and profiling MPI+OpenMP**

- Summary
Thread Correctness – Intel ThreadChecker

- Intel ThreadChecker operates in a similar fashion to helgrind,
- Compile with `-tcheck`, then run program using `tcheck_cl`:

Application finished

<table>
<thead>
<tr>
<th>ID</th>
<th>Short Description</th>
<th>Severity</th>
<th>Context</th>
<th>Description</th>
<th>1st Acc</th>
<th>2nd Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Write -&gt; Error</td>
<td>Error</td>
<td>pthread</td>
<td>Memory write of global_variable at &quot;pthread_race.c&quot;:31 conflicts with</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Write data</td>
<td>Normal</td>
<td>pthread</td>
<td>&quot;pthread_race.c&quot;:31 conflicts with</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ta-race</td>
<td>Normal</td>
<td>e.c</td>
<td>a prior memory write of</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>global_variable at</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&quot;pthread_race.c&quot;:31 (output</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dependence)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thread Correctness – Intel ThreadChecker  2/3

- One may output to HTML:

  
  ```
  tcheck_cl --format HTML --report pthread_race.html pthread_race
  ```

![Thread Checker Output](image)

<table>
<thead>
<tr>
<th>ID</th>
<th>Short Description</th>
<th>Severity Name</th>
<th>Count</th>
<th>ContextResult</th>
<th>Description</th>
<th>1st AccessResult</th>
<th>2nd AccessResult</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write =&gt; data race</td>
<td>Error</td>
<td>1</td>
<td></td>
<td><em>pthread_race.c:25</em> Memory write of global variable at <em>pthread_race.c:31</em> conflicts with a prior memory write of global variable at <em>pthread_race.c:31</em> (output dependent)</td>
<td><em>pthread_race.c:31</em></td>
<td><em>pthread_race.c:31</em></td>
</tr>
<tr>
<td>2</td>
<td>Thread termination</td>
<td>Information</td>
<td>1</td>
<td>Whole Program 1</td>
<td><em>pthread_race.c:43</em> Thread termination at <em>pthread_race.c:43</em> - includes stack allocation of 8,094 MB and use of 4,472 KB</td>
<td><em>pthread_race.c:43</em></td>
<td><em>pthread_race.c:43</em></td>
</tr>
<tr>
<td>3</td>
<td>Thread termination</td>
<td>Information</td>
<td>1</td>
<td>Whole Program 2</td>
<td><em>pthread_race.c:43</em> Thread termination at <em>pthread_race.c:43</em> - includes stack allocation of 8,094 MB and use of 4,472 KB</td>
<td><em>pthread_race.c:43</em></td>
<td><em>pthread_race.c:43</em></td>
</tr>
<tr>
<td>4</td>
<td>Thread termination</td>
<td>Information</td>
<td>1</td>
<td>Whole Program 3</td>
<td><em>pthread_race.c:43</em> Thread termination at <em>pthread_race.c:43</em> - includes stack allocation of 8 MB and use of 4 KB</td>
<td><em>pthread_race.c:43</em></td>
<td><em>pthread_race.c:43</em></td>
</tr>
</tbody>
</table>

Courtesy of Rainer Keller, HLRS and ORNL
Thread Correctness – Intel ThreadChecker 3/3

• If one wants to compile with threaded Open MPI (option for IB):

\[
\text{configure --enable-mpi-threads} \\
\text{--enable-debug} \\
\text{--enable-mca-no-build=memory-ptmalloc2} \\
\text{CC=icc F77=ifort FC=ifort} \\
\text{CFLAGS='--debug all --inline-debug-info tcheck'} \\
\text{CXXFLAGS='--debug all --inline-debug-info tcheck'} \\
\text{FFLAGS='--debug all -tcheck' LDFLAGS='tcheck'}
\]

• Then run with:

\[
\text{mpirun --mca tcp,sm,self -np 2 tcheck_cl} \\
\text{--reinstrument -u full --format html} \\
\text{--cache_dir '/tmp/my_username_${}_tc_cl_cache'} \\
\text{--report 'tc_mpi_test_suite_${}'} \\
\text{--options 'file=tc_my_executable_%H_%I,} \\
\text{pad=128, delay=2, stall=2' --} \\
\text{./my_executable my_arg1 my_arg2 ...}
\]
Performance Tools Support for Hybrid Code

- Paraver examples have already been shown, tracing is done with linking against (closed-source) `omptrace` or `ompitrace`.

- For Vampir/Vampirtrace performance analysis:
  ```
  ./configure --enable-omp
  --enable-hyb
  --with-mpi-dir=/opt/OpenMPI/1.3-icc
  CC=icc F77=ifort FC=ifort
  (Attention: does not wrap MPI_Init_thread!)
  ```

Courtesy of Rainer Keller, HLRS and ORNL
Kojak – Example “Wait at Barrier”

Indication of non-optimal load balance

Screenshots, courtesy of KOJAK JSC, FZ Jülich
Kojak – Example “Wait at Barrier”, Solution

Better load balancing with dynamic loop schedule
Outline

• Introduction / Motivation
• Programming models on clusters of SMP nodes
• Case Studies / pure MPI vs hybrid MPI+OpenMP
• Practical “How-To” on hybrid programming
• Mismatch Problems
• Opportunities:
  Application categories that can benefit from hybrid parallelization
• Thread-safety quality of MPI libraries
• Tools for debugging and profiling MPI+OpenMP

• Summary
Acknowledgements

- We want to thank
  - Gerhard Wellein, RRZE
  - Rainer Keller, HLRS and ORNL
  - Jim Cownie, Intel
  - KOJAK project at JSC, Research Center Jülich
  - HPCMO Program and the Engineer Research and Development Center Major Shared Resource Center, Vicksburg, MS (http://www.erdc.hpc.mil/index)
Summary – the good news

MPI + OpenMP

• Significant opportunity → higher performance on smaller number of threads
• Seen with NPB-MZ examples
  – BT-MZ → strong improvement (as expected)
  – SP-MZ → small improvement (none was expected)
• Usable on higher number of cores
• Advantages
  – Load balancing
  – Memory consumption
  – Two levels of parallelism
    • Outer → distributed memory → halo data transfer → MPI
    • Inner → shared memory → ease of SMP parallelization → OpenMP
• You can do it → “How To”
Summary – the bad news

MPI+OpenMP: There is a huge amount of pitfalls

• Pitfalls of MPI
• Pitfalls of OpenMP
  – On ccNUMA → e.g., first touch
  – Pinning of threads on cores
• Pitfalls through combination of MPI & OpenMP
  – E.g., topology and mapping problems
  – Many mismatch problems
• Tools are available 😊
  – It is not easier than analyzing pure MPI programs 😞
• Most hybrid programs → Masteronly style
• Overlapping communication and computation with several threads
  – Requires thread-safety quality of MPI library
  – Loss of OpenMP support → future OpenMP subteam concept
Summary – good and bad

• Problems may be small
  – $x\%$ loss efficiency $\xrightarrow{\text{mismatch}} f \times x\%$ loss
  – If loss is small $x=1\%$
    and factor $f=3$ is medium
    $\Rightarrow$ don’t worry ?!

• Optimization
  – 1 MPI process per core $\times \cdots \times$ per SMP node
    $\wedge$ somewhere between
    may be the optimum

• Efficiency of MPI+OpenMP is not for free:
  The efficiency strongly depends on
  the amount of work in the source code development
Summary – Alternatives

Pure MPI
+ Ease of use
  – Topology and mapping problems may need to be solved
    (depends on loss of efficiency with these problems)
  – Number of cores may be more limited than with MPI+OpenMP
+ Good candidate for perfectly load-balanced applications

Pure OpenMP
+ Ease of use
  – Limited to problems with tiny communication footprint
  – source code modifications are necessary
    (Variables that are used with “shared” data scope
    must be allocated as “sharable”)
± (Only) for the appropriate application a suitable tool
Summary

• This tutorial tried to
  – help to negotiate obstacles with hybrid parallelization,
  – give hints for the design of a hybrid parallelization,
  – and technical hints for the implementation → “How To”,
  – show tools if the application does not work as designed.

• This tutorial was not an introduction into other parallelization models:
  – Partitioned Global Address Space (PGAS) languages (Unified Parallel C (UPC), Co-array Fortran (CAF), Chapel, Fortress, Titanium, and X10).
  – High Performance Fortran (HPF)
  → Many rocks in the cluster-of-SMP-sea do not vanish into thin air by using new parallelization models
  → Area of interesting research in next years
Conclusions

• Future hardware will be more complicated
  – Heterogeneous
  – ccNUMA quality may be lost on cluster nodes
  – ....

• High-end programming → more complex

• Medium number of cores → more simple
  (if \#cores / SMP-node will not shrink)

• MPI+OpenMP → work horse on large systems

• Pure MPI → still on smaller cluster

• OpenMP → on large ccNUMA nodes
  (not ClusterOpenMP)

Thank you for your interest

Q & A

Please fill in the feedback sheet – Thank you
Appendix

- Abstract
- Authors
- References (with direct relation to the content of this tutorial)
- Further references
Abstract

Half-Day Tutorial  (Level: 25% Introductory, 50% Intermediate, 25% Advanced)

Authors.  Rolf Rabenseifner, HLRS, University of Stuttgart, Germany
Georg Hager, University of Erlangen-Nuremberg, Germany
Gabriele Jost, Texas Advanced Computing Center / Naval Postgraduate School, USA

Abstract.  Most HPC systems are clusters of shared memory nodes. Such systems can be PC clusters with dual or quad boards and single or multi-core CPUs, but also "constellation" type systems with large SMP nodes. Parallel programming may combine the distributed memory parallelization on the node inter-connect with the shared memory parallelization inside of each node.

This tutorial analyzes the strength and weakness of several parallel programming models on clusters of SMP nodes. Various hybrid MPI+OpenMP programming models are compared with pure MPI. Benchmark results of several platforms are presented. The thread-safety quality of several existing MPI libraries is also discussed. Case studies will be provided to demonstrate various aspects of hybrid MPI/OpenMP programming. Another option is the use of distributed virtual shared-memory technologies. Application categories that can take advantage of hybrid programming are identified. Multi-socket-multi-core systems in highly parallel environments are given special consideration.

Dr. Rolf Rabenseifner studied mathematics and physics at the University of Stuttgart. Since 1984, he has worked at the High-Performance Computing-Center Stuttgart (HLRS). He led the projects DFN-RPC, a remote procedure call tool, and MPI-GLUE, the first metacomputing MPI combining different vendor's MPIs without losing the full MPI interface. In his dissertation, he developed a controlled logical clock as global time for trace-based profiling of parallel and distributed applications. Since 1996, he has been a member of the MPI-2 Forum and since Dec. 2007, he is in the steering committee of the MPI-3 Forum. From January to April 1999, he was an invited researcher at the Center for High-Performance Computing at Dresden University of Technology. Currently, he is head of Parallel Computing - Training and Application Services at HLRS. He is involved in MPI profiling and benchmarking, e.g., in the HPC Challenge Benchmark Suite. In recent projects, he studied parallel I/O, parallel programming models for clusters of SMP nodes, and optimization of MPI collective routines. In workshops and summer schools, he teaches parallel programming models in many universities and labs in Germany.
Dr. Georg Hager studied theoretical physics at the University of Bayreuth, specializing in nonlinear dynamics and holds a PhD in Computational Physics from the University of Greifswald. Since 2000 he is a member of the HPC Services group at the Regional Computing Center Erlangen (RRZE), which is part of the University of Erlangen-Nuremberg. His daily work encompasses all aspects of user support in High Performance Computing like tutorials and training, code parallelization, profiling and optimization and the assessment of novel computer architectures and tools. Recent research includes architecture-specific optimization strategies for current microprocessors and special topics in shared memory programming.
Gabriele Jost obtained her doctorate in Applied Mathematics from the University of Göttingen, Germany. For more than a decade she worked for various vendors (Suprenum GmbH, Thinking Machines Corporation, and NEC) of high performance parallel computers in the areas of vectorization, parallelization, performance analysis and optimization of scientific and engineering applications.

In 1998 she joined the NASA Ames Research Center in Moffett Field, California, USA as a Research Scientist. Here her work focused on evaluating and enhancing tools for parallel program development and investigating the usefulness of different parallel programming paradigms.

In 2005 she moved from California to the Pacific Northwest and joined Sun Microsystems as a staff engineer in the Compiler Performance Engineering team. Her task is the analysis of compiler generated code and providing feedback and suggestions for improvement to the compiler group. Her research interest remains in area of performance analysis and evaluation of programming paradigms for high performance computing. Currently, she is working at the Texas Advanced Computing Center / Naval Postgraduate School.
References (with direct relation to the content of this tutorial)

- **NAS Parallel Benchmarks:**
  http://www.nas.nasa.gov/Resources/Software/npb.html

- R.v.d. Wijngaart and H. Jin,
  *NAS Parallel Benchmarks, Multi-Zone Versions*,
  NAS Technical Report NAS-03-010, 2003

- H. Jin and R. v.d.Wijngaart,
  *Performance Characteristics of the multi-zone NAS Parallel Benchmarks*,
  Proceedings IPDPS 2004

- G. Jost, H. Jin, D. an Mey and F. Hatay,
  *Comparing OpenMP, MPI, and Hybrid Programming*,
  Proc. Of the 5th European Workshop on OpenMP, 2003

- E. Ayguade, M. Gonzalez, X. Martorell, and G. Jost,
  *Employing Nested OpenMP for the Parallelization of Multi-Zone CFD Applications*,
  Proc. Of IPDPS 2004
References

• Rolf Rabenseifner,
**Hybrid Parallel Programming on HPC Platforms.**
In proceedings of the Fifth European Workshop on OpenMP, EWOMP ’03,

• Rolf Rabenseifner,
**Comparison of Parallel Programming Models on Clusters of SMP Nodes.**
In proceedings of the 45nd Cray User Group Conference, CUG SUMMIT 2003,
May 12-16, Columbus, Ohio, USA.

• Rolf Rabenseifner and Gerhard Wellein,
**Comparison of Parallel Programming Models on Clusters of SMP Nodes.**
In Modelling, Simulation and Optimization of Complex Processes (Proceedings of
the International Conference on High Performance Scientific Computing,

• Rolf Rabenseifner and Gerhard Wellein,
**Communication and Optimization Aspects of Parallel Programming Models
on Hybrid Architectures.**
In the **International Journal of High Performance Computing Applications,**
References

- Rolf Rabenseifner, 
  **Communication and Optimization Aspects on Hybrid Architectures.**

- Rolf Rabenseifner and Gerhard Wellein, 
  **Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures.**
  In proceedings of the Fourth European Workshop on OpenMP (EWOMP 2002), Roma, Italy, Sep. 18-20th, 2002.

- Rolf Rabenseifner, 
  **Communication Bandwidth of Parallel Programming Models on Hybrid Architectures.**
References


Further references

- Sergio Briguglio, Beniamino Di Martino, Giuliana Fogaccia and Gregorio Vlad, *Hierarchical MPI+OpenMP implementation of parallel PIC applications on clusters of Symmetric MultiProcessors*, 10th European PVM/MPI Users' Group Conference (EuroPVM/MPI'03), Venice, Italy, 29 Sep - 2 Oct, 2003

- Barbara Chapman, *Parallel Application Development with the Hybrid MPI+OpenMP Programming Model*, Tutorial, 9th EuroPVM/MPI & 4th DAPSYS Conference, Johannes Kepler University Linz, Austria September 29-October 02, 2002


Further references

- Holger Brunst and Bernd Mohr,
  **Performance Analysis of Large-scale OpenMP and Hybrid MPI/OpenMP Applications with VampirNG**
  http://www.fz-juelich.de/zam/kojak/documentation/publications/

- Felix Wolf and Bernd Mohr,
  **Automatic performance analysis of hybrid MPI/OpenMP applications**
  http://www.fz-juelich.de/zam/kojak/documentation/publications/

- Felix Wolf and Bernd Mohr,
  **Automatic Performance Analysis of Hybrid MPI/OpenMP Applications**
  http://www.fz-juelich.de/zam/kojak/documentation/publications/
Further references


Further references


Further references


Further references


Content

• Introduction / Motivation ........................................ 1
• Programming models on clusters of SMP nodes .......... 6
  – Major programming models 7
  – Pure MPI 9
  – Hybrid Masteronly Style 10
  – Overlapping Communication and Computation 11
  – Pure OpenMP 12
• Case Studies / pure MPI vs. hybrid MPI+OpenMP ........ 13
  – The Multi-Zone NAS Parallel Benchmarks 14
  – Benchmark Architectures 17
  – On the Sun Constellation Cluster Ranger 19
  – NUMA Control (numactl) 24
  – On a Cray XT5 cluster 30
  – On a Cray XT4 cluster 35
  – On a IBM Power6 system 39
  – Conclusions 42
• Practical “How-To” on hybrid programming ............... 43
  – How to compile, link and run 45
  – Running the code efficiently? 52
  – A short introduction to ccNUMA 54
  – ccNUMA Memory Locality Problems / First Touch 58
  – ccNUMA problems beyond first touch 61
  – Bandwidth and latency 63
  – OpenMP overhead 66
  – Thread/Process Affinity (“Pinning”) 68
  – Example: 3D Jacobi Solver 71
  – Example: Sparse Matrix-Vector-Multiply with JDS 74
• Mismatch Problems .............................................. 80
  – Topology problem 82
  – Mapping problem with mixed model 88
  – Unnecessary intra-node communication 89
  – Sleeping threads and network saturation problem 90
  – Additional OpenMP overhead 91
  – Overlapping communication and computation 92
  – Communication overhead with DSM 96
  – Back to the mixed model 101
  – No silver bullet 102
• Opportunities: Application categories that can benefit from hybrid parallelization 103
  – Nested Parallelism 104
  – Load-Balancing 105
  – Memory consumption 106
  – Opportunities, if MPI speedup is limited due to algorithmic problem 109
  – To overcome MPI scaling problems 110
  – Summary 101
Content

- Thread-safety quality of MPI libraries  112
  - MPI rules with OpenMP  114
  - Thread support of MPI libraries  117
  - Thread Support within OpenMPI  118
- Tools for debugging and profiling MPI+OpenMP  119
  - Intel ThreadChecker  120
  - Performance Tools Support for Hybrid Code  123
- Summary  126
  - Acknowledgements  127
  - Summaries  128
  - Conclusions  133

- Appendix  134
  - Abstract  135
  - Authors  136
  - References (with direct relation to the content of this tutorial)  139
  - Further references  143
- Content  149