Hybrid MPI & OpenMP Parallel Programming

MPI + OpenMP and other models on clusters of SMP nodes

Rolf Rabenseifner\(^1\) \hspace{1cm} Georg Hager\(^2\) \hspace{1cm} Gabriele Jost\(^3\)

\(^1\) High Performance Computing Center (HLRS), University of Stuttgart, Germany
\(^2\) Regional Computing Center (RRZE), University of Erlangen, Germany
\(^3\) Supersmith, Maximum Performance Software, USA

Tutorial tut163 at SC12, November 11, 2012, Salt Lake City, Utah, USA
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
- Other options on clusters of SMP nodes
- Summary
- Appendix
- Content (detailed)
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
  - Other options on clusters of SMP nodes
- 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
  - local data in each process

**OpenMP**
- (shared data)
  - some_serial_code
  - #pragma omp parallel for
    - for (j=...;...; j++)
      - block_to_be_parallelized
    - again_some_serial_code

**Node Interconnect**
- MPI between the nodes via node interconnect

**SMP nodes**
- OpenMP inside of the
  - 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

Masteronly
MPI only outside of parallel regions

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

Major Problems

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

→ Section Thread-safety quality of MPI libraries
Overlapping Communication and Computation

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

```c
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)

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

Experience: Mismatch section
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 (Supersmith, Maximum Performance Software)

• 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
• Other options on clusters of SMP nodes
• 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>

- Diagram:
  - Set up zones
  - Initialize
  - Exchange boundaries
  - Solve LU/SP/BT
  - Verify

- Table:
  - Time step: sequential, sequential, sequential
  - Inter-zones: MPI Processes, MLP Processes, OpenMP
  - Exchange boundaries: Call MPI, data copy+ sync., OpenMP
  - Intra-zones: OpenMP, OpenMP, OpenMP
call omp_set_numthreads (weight)
do step = 1, itmax
   call exch_qbc(u, qbc, nx,...)

   call mpi_send/recv

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

end do

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

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

   do k = 2, nz-1
      !$OMP DO
         do j = 2, ny-1
            do i = 2, nx-1
               do m = 1, 5
                  u(m,i,j,k) = dt*rsd(m,i,j,k-1)
               end do
            end do
         end do
      !$OMP END DO NOWAIT
   end do

   !$OMP END PARALLEL

subroutine zsolve(u, rsd,...)
call omp_set_numthreads (weight)
do step = 1, itmax
    call exch_qbc(u, qbc, nx,...)

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

...
Pipelined Thread Execution in SSOR

subroutine ssor
  !$OMP PARALLEL DEFAULT(SHARED)
  !$OMP& PRIVATE(m, i, j, k...)
  call sync1()
  do k = 2, nz-1
    !$OMP DO
      do j = 2, ny-1
        do i = 2, nx-1
          do m = 1, 5
            rsd(m, i, j, k) =
            dt*rsd(m, i, j, k-1) + ...
          end do
        end do
      end do
    !$OMP END DO nowait
  end do
  call sync2()
  ...
  !$OMP END PARALLEL
  ...

subroutine sync1
  ...neigh = iam -1
  do while (isync(neigh) .eq. 0)
    !$OMP FLUSH(isync)
  end do
  isync(neigh) = 0
  !$OMP FLUSH(isync)
  ...

subroutine sync2
  ...neigh = iam -1
  do while (isync(neigh) .eq. 1)
    !$OMP FLUSH(isync)
  end do
  isync(neigh) = 1
  !$OMP FLUSH(isync)
  ...
Golden Rule for ccNUMA: “First touch”

• A memory page gets mapped into the local memory of the processor that first touches it!
• "touch" means "write", not "allocate"

```c
do iz = 1, proc_num_zones
  zone = proc_zone_id(iz)
call adi(rho_i(start1(iz)), us(start1(iz)),
  vs(start1(iz)), ws(start1(iz))
...
$ end do
$ end do
```

All benchmarks use first touch policy to achieve good memory placement!
Benchmark Characteristics

- Aggregate sizes:
  - 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)
  - Alternative Directions Implicit (ADI) method
  - #Zones: 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)
  - SSOR method (2D pipelined method)
  - #Zones: 16 (all Classes)
  - Size of the zones identical:
    - no load-balancing required
    - limited parallelism on outer level

- **SP-MZ:** (Scalar Pentadiagonal simulated CFD application)
  - #Zones: 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
Hybrid code on cc-NUMA architectures

- **OpenMP**:
  - Support only per MPI process
  - Version 3.1 has support for binding of threads via OMP_PROC_BIND environment variable.
  - Under consideration for Version 4.0: OMP_PROC_SET or OMP_LIST to restrict the execution to a subset of the machine; OMP_AFFINITY to influence how the threads are distributed and bound on the machine
  - **Version 4.0 announced at SC12**

- **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,taskset,dplace,omplace etc`
  - See [http://www.halobates.de/numaapi3.pdf](http://www.halobates.de/numaapi3.pdf)
  - More in “How-to’s”
Benchmark Architectures

- Dell Linux Cluster Lonestar
- Cray XE6: Hector/Hermit
- IBM Power 6
Dell Linux Cluster Lonestar

- Located at the Texas Advanced Computing Center (TACC), University of Texas at Austin (http://www.tacc.utexas.edu)
- 1888 nodes, 2 Xeon Intel 6-Core 64-bit Westmere processors, 3.33 GHz, 24 GB memory per node, Peak Performance 160 Gflops per node, 3 channels from each processor's memory controller to 3 DDR3 ECC DIMMS, 1333 MHz,
- Processor interconnect, QPI, 6.4GT/s
- Node Interconnect: InfiniBand Mellanox Switches, fat-tree topology, 40Gbit/sec point-to-point bandwidth
- More details: http://www.tacc.utexas.edu/user-services/user-guides/lonestar-user-guide
- Compiling the benchmarks: I
  - fort 11.1, Options: -O3 –ipo –openmp –mcmodel=medium
- Running the benchmarks:
  - MVAPICH 2
  - setenv OMP_NUM_THREADS=
  - ibrun tacc_affinity ./bt-mz.x
Example run script

```bash
#!/bin/csh
#$ -cwd
#$ -j y
#$ -q systest
#$ -pe 12way 24
#$ -V
#$ -l h_rt=00:10:00
setenv OMP_NUM_THREADS 1
setenv MY_NSLOTS 16
ibrun tacc_affinity ./bin/sp-mz.D.
```

- Run 12 MPI processes per node, allocate 24 cores (2 nodes) altogether
- 1 thread per MPI process
- Only use 16 of the 24 cores for MPI.
  NOTE: 8 cores unused!!!
- Command to run mpi job
- numactl script for process/thread placement
## NUMA Operations

<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>-c</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>-l</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} select only one</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>
NUMA Operations: Memory Placement

Memory allocation:
- **MPI**
  - Pure MPI: socket local allocation is best
  - Hybrid: Depending on #threads per process remote socket memory may be required
- **OpenMP**
  - *Regular structured access pattern that does not change*: Allocate close to core where thread runs
  - *Irregular, unpredictable access*: Round-robin placement of pages
- Once allocated, a memory-structure is fixed

Example: `numactl -c 1 -l ./a.out`
Use socket 1, allocate memory on current socket
Example numactl script

```bash
myway=`echo $PE | sed s/way//`
export MV2_USE_AFFINITY=0
export MV2_ENABLE_AFFINITY=0
my_rank=$PMI_RANK
local_rank=$(( my_rank % myway ))
if [ $myway -eq 12 ]; then
    numnode=$(( local_rank / 6 ))
fi
exec numactl -c $numnode -m $numnode $*
```
### Dell Linux Cluster Lonestar Topology

<table>
<thead>
<tr>
<th>Socket 0:</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>9</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>32kB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256kB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12MB</td>
</tr>
<tr>
<td>Socket 1:</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>32kB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256kB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12MB</td>
</tr>
</tbody>
</table>
Dell Linux Cluster Lonestar Topology

CPU type: Intel Core Westmere processor

Hardware Thread Topology

Socket 0: ( 1 3 5 7 9 11 )
Socket 1: ( 0 2 4 6 8 10 )

Careful! Numbering scheme of cores is system dependent
No idle cores

BT-MZ improves using hybrid as expected due to better load balance

Unexpected: SP-MZ improves in some cases using hybrid
Pitfall (1): Running 2 threads on the same core

Running NPB BT-MZ Class D 128 MPI Procs, 12 threads each, 1 MPI per node (1way)

Pinning A:
exec numactl -c 0 -m 0 $*

Only use cores and memory on socket 0, 12 threads on 6 cores

Running 128 MPI Procs, 12 threads each
Pinning B:
exec numactl -c 0,1 -m 0,1 $*

Use cores and memory on 2 sockets
Pitfall (2): Cause remote memory access

Running NPB BT-MZ Class D 128 MPI Procs, 6 threads each 2 MPI per node

Pinning A:
if [ $localrank == 0 ]; then
  exec numactl --physcpubind=0,1,2,3,4,5 -m 0 $*
elif [ $localrank == 1 ]; then
  exec numactl --physcpubind=6,7,8,9,10,11 -m 1 $*
fi

Running 128 MPI Procs, 6 threads each

Pinning B:
if [ $localrank == 0 ]; then
  exec numactl --physcpubind=0,2,4,6,8,10 -m 0 $*
elif [ $localrank == 1 ]; then
  exec numactl --physcpubind=1,3,5,7,9,11 -m 1 $*
fi

Half of the threads access remote memory

600 Gflops

900 Gflops

Only local memory access
• LU-MZ significantly benefits from hybrid mode:
  - Pure MPI limited to 16 cores, due to \#zones = 16
• Decrease of resource contention large contribution to improvement
Cray XE6 Hermit

- Located at HLRS Stuttgart, Germany ([https://wickie.hlrs.de/platforms/index.php/Cray_XE6](https://wickie.hlrs.de/platforms/index.php/Cray_XE6))
- 3552 compute nodes 113,664 cores
- Each node contains two AMD 6276 Interlagos processor with 16 cores each, running at 2.3 GHz (TurboCore 3.3GHz)
- Around 1 Pflop theoretical peak performance
- 32 GB of main memory available per node
- 32-way shared memory system
- High-bandwidth interconnect using Cray Gemini communication chips.
Cray XE6 Hermit Topology

CPU type: AMD Interlagos processor

Hardware Thread Topology

Sockets: 2
Cores per socket: 16
Threads per core: 1

Socket 0: (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
Socket 1: (16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31)

4 NUMA Domains per Node
Cray XE6 Hermit Scalability

NPB-MZ class E on Hermit

SUPER

MPI/OpenMP

GFlops

2000
4000
6000
8000
10000
12000

2000
4000
6000
8000
10000
12000

32K cores
16K cores
8K cores

Rabenseifner, Hager, Jost
Hybrid Parallel Programming
Expected:
BT-MZ benefits from hybrid approach:
- high number of MPI processes yields bad workload distribution
- Best MPIxOMP combination depends on problem size

Expected:
- Both benchmarks benefit by increasing parallelism

Unexpected:
SP-MZ improves when reducing number of MPI processes
BT-MZ 1024x32 unexpected low performance
Cray XE6 Hector

- Located at EPCC, Edinburgh, Scotland, UK National Supercomputing Services, Hector Phase 2b (http://www.hector.ac.uk)
- 1856 XE6 compute nodes.
- Each node contains two AMD 2.1 GHz 12-core processors giving a total of 44,544 cores
- Around 373 Tflops theoretical peak performance
- 32 GB of main memory available per node
- 24-way shared memory system.
- High-bandwidth interconnect using Cray Gemini communication chips.

CPU type: AMD Magny Cours processor
Hardware Thread Topology
Sockets: 2
Cores per socket: 12
Threads per core: 1

---

no SMT
Cray XE6 Hector Node Topology

Graphical:

Socket 0:

- 0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11
- 64kB - 64kB - 64kB - 64kB - 54kB - 64kB - 64kB - 64kB - 64kB - 64kB - 64kB - 64kB
- 512kB - 512kB - 512kB - 512kB - 512kB - 512kB - 512kB - 512kB - 512kB - 512kB - 512kB - 512kB
- 5MB

Socket 1:

- 64kB - 64kB - 64kB - 64kB - 54kB - 64kB - 64kB - 64kB - 64kB - 64kB - 64kB - 64kB
- 512kB - 512kB - 512kB - 512kB - 512kB - 512kB - 512kB - 512kB - 512kB - 512kB - 512kB - 512kB
- 5MB

4 NUMA domains
BT-MZ improves using hybrid as expected due to better load balance…

BUT: …unexpected low performance for 4096 cores..?

1 SMP node = 2 AMD Magny Cours = 4 NUMA domains = 24 cores
Craypat BT-MZ 256x16

Number of PEs (MPI ranks): 256

Numbers of PEs per Node: 1 PE on each of 256 Nodes

Numbers of Threads per PE: 16 threads on each of 248 PEs

Numbers of Threads per PE: 17 threads on each of 8 PEs

Number of Cores per Socket: 12

export NPB_MZ_BLOAD=0
Benchmark will not try to load-balance between threads

Number of PEs (MPI ranks): 256

Numbers of PEs per Node: 1 PE on each of 256 Nodes

Numbers of Threads per PE: 16

Number of Cores per Socket: 12

Benchmark tries to balance load, aprun –d 16 yields multiple threads on same core!
BT-MZ improves using hybrid as expected due to better load balance...

Unexpected: SP-MZ improves in some cases using hybrid
Cray XE6: CrayPat Performance Analysis

- module load xt-craypat
- Compilation:
  - ftn –fastsse –r8 –mp[= trace ]
- Instrument:
  - pat_build –w –g mpi,omp bt.exe bt.exe.pat
- Execution:
  - (export PAT_RT_HWPC {0,1,2,..})
  - export OMP_NUM_THREADS 4
  - aprun –n NPROCS –d 4 ./bt.exe.pat
- Generate report:
  - pat_report –O
    load_balance,thread_times,program_time,mpi_callers –O
    profile_pe.th $1

-d depth Specifies the number of CPUs for each PE and its threads.
BT-MZ 32x4 Function Profile

```fortran
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(n,m,k,i,j,ksize)
!$OMP& SHARED(dz5,dz4,dz3,dz2,dz1,tz2,tz1,dt,c1345,c4,c3,con43,c3c4,c1,nt=1)
!$OMP& c2,nx,ny,nz)
  ksize = nz-1
  
  c Compute the indices for storing the block-diagonal matrix;
  c determine c (labeled f) and s jacobians
  
  !$OMP DO
    do j = 1, ny-2
      do i = 1, nx-2
        do k = 0, ksize
          tmp1 = 1.d0 / u(1,i,j,k)
          tmp2 = tmp1 * tmp1
          tmp3 = tmp1 * tmp2
          fjac(1,1,k) = 0.d0
          fjac(1,2,k) = 0.d0
          fjac(1,3,k) = 0.d0
          fjac(1,4,k) = 1.d0
          fjac(1,5,k) = 0.d0
        
```

Hybrid Slide
BT-MZ Load-Balance 32x4 vs 128x1

Table 2: Load Balance across PE's by Function Group

<table>
<thead>
<tr>
<th>Time %</th>
<th>Time</th>
<th>Calls</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0%</td>
<td>1,782</td>
<td>608</td>
<td>Thread</td>
</tr>
<tr>
<td>86.1%</td>
<td>1,535</td>
<td>618</td>
<td>thread,1</td>
</tr>
<tr>
<td>77.8%</td>
<td>1,535</td>
<td>618</td>
<td>thread,2</td>
</tr>
<tr>
<td>68.1%</td>
<td>1,535</td>
<td>618</td>
<td>thread,3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bt-mz-C.32x4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7%</td>
</tr>
<tr>
<td>2.7%</td>
</tr>
<tr>
<td>2.7%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bt-mz-C.128x1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5%</td>
</tr>
<tr>
<td>0.5%</td>
</tr>
<tr>
<td>0.0%</td>
</tr>
</tbody>
</table>

| 44.9% | 10,894 | 17983 | USER |
| 0.7% | 23,205 | 9093 | thread,0 |
| 0.3% | 10,084 | 26873 | thread,110 |
| 0.3% | 8,070 | 17983 | thread,91 |

- 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
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!
- 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
LU-MZ significantly benefits from hybrid mode:

- Pure MPI limited to 16 cores, due to #zones = 16
Conclusions:

- **BT-MZ:**
  - Inherent workload imbalance on MPI level
  - \#nprocs = \#nzones yields poor performance
  - \#nprocs < \#zones => better workload balance, but decreases parallelism
  - Hybrid MPI/OpenMP yields better load-balance, maintains amount of parallelism

- **SP-MZ:**
  - No workload imbalance on MPI level, pure MPI should perform best
  - MPI/OpenMP outperforms MPI on some platforms due contention to network access within a node

- **LU-MZ:**
  - Hybrid MPI/OpenMP increases level of parallelism

- **All Benchmarks:**
  - Decrease network pressure
  - Lower memory requirements
  - Good process/thread affinity 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
• Other options on clusters of SMP nodes
• 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
    - Other overhead
  - 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, -fopenmp, -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](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
Some examples for compilation and execution (2)

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
      ```
      test -s ~/myexports && . ~/myexports
      ```
    in your ~/.bashrc
    
    • Add
      ```
      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!
Some examples for compilation and execution (3)

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 (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 or similar mechanisms

- 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

Examples with mpiexec

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

  $ export OMP_NUM_THREADS=8
  $ mpiexec -pernode ./a.out

- Same but 2 MPI processes per node:

  $ export OMP_NUM_THREADS=4
  $ mpiexec -npernode 2 ./a.out

- Pure MPI:

  $ export OMP_NUM_THREADS=1 # or nothing if serial code
  $ mpiexec ./a.out

Where do the threads run?
→ see later!
Running the code **efficiently**?

- Symmetric, UMA-type compute nodes have become rare animals
  - NEC SX
  - Intel 1-socket (“Port Townsend/Melstone/Lynnfield”) – 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), Intel Sandy Bridge)

  - 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 access across locality domains
  – 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 *transparency accessible* by all processors
  - but *physically distributed*
  - with *varying bandwidth and latency*
  - and *potential contention* (shared memory paths)
How much does non-local access cost?

- Example: AMD Magny Cours 2-socket system (4 chips, 2 sockets)
  
  *STREAM bandwidth measurements*
**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
"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
Most simple case: explicit initialization

```fortran
integer, parameter :: N = 10000000
double precision A(N), B(N)
A = 0.d0
!
OMP parallel do
  do i = 1, N
    B(i) = function ( A(i) )
  end do
!OMP end parallel do
```

```fortran
integer, parameter :: N = 10000000
double precision A(N), B(N)
!
OMP parallel
equiv
!
OMP do schedule(static)
do i = 1, N
  A(i) = 0.d0
end do
!
OMP end do
equiv
!
OMP do schedule(static)
do i = 1, N
  B(i) = function ( A(i) )
end do
!
OMP end parallel do
```
**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 array copy loops (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):

```fortran
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
enddo

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)

- inter-node
- inter-socket
- revolving buffers
- intra-socket

Shared cache advantage

- Between two cores of one socket
- Between two sockets of one node
- Between two nodes via InfiniBand

Affinity matters!
The parallel vector triad benchmark
A "swiss army knife" for microbenchmarking

• What about OpenMP overhead?
• Simple streaming benchmark:

```c
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);
}
```

• Report performance for different N
• Choose NITER so that accurate time measurement is possible
• Triad results lead to a deep understanding of multicore architecture and OpenMP performance overhead
The parallel vector triad benchmark
Optimal code on x86 machines

```c
#include <vector>
#include <omp.h>

int main() {
    int niter = 1000000;
    int size = vector_size(niter);

    timing(&wct_start, &cput_start);
    #pragma omp parallel private(j)
    {
        for(j=0; j<niter; j++)
            if(size > CACHE_SIZE>>5) {
                #pragma omp parallel for
                #pragma vector always
                #pragma vector aligned
                #pragma vector non-temporal
                for(i=0; i<size; ++i)
                    a[i]=b[i]+c[i]*d[i];
            } else {
                #pragma omp parallel for
                #pragma vector always
                #pragma vector aligned
                for(i=0; i<size; ++i)
                    a[i]=b[i]+c[i]*d[i];
            }
        if(a[5]<0.0)
    }
    timing(&wct_end, &cput_end);
    return 0;
}
```

Large-N version (NT)

```c
int vector_size(int n) {
    return int(pow(1.3, n)) & (-8);
}
```

Small-N version (noNT)
The parallel vector triad benchmark

Single thread on AMD Interlagos chip

OMP overhead and/or lower optimization w/ OpenMP active

Team restart is expensive!

use only outer parallel from now on!
The parallel vector triad benchmark

Intra-chip scaling on Interlagos chip

- sync overhead
- Per-module L2 caches
- Aggregate L2, exclusive L3
- L2 bottleneck
- Memory BW saturated @ 4 threads
The parallel vector triad benchmark

*Nontemporal stores on Interlagos chip*

- NT stores hazardous if data in cache
- 25% speedup for vector triad in memory via NT stores

![Graph showing performance vs. loop length N for OpenMP T=8 and OpenMP T=8 NT.](image)
The parallel vector triad benchmark

Topology dependence on Interlagos chip

- sync overhead nearly topology-independent
- more aggregate L3 with more chips
- bandwidth scalability across memory interfaces
The parallel vector triad benchmark

Inter-chip scaling on Interlagos node
Most of the OpenMP overhead is barrier sync!

But how much is it exactly, and does it depend on the topology?

Overhead in cycles:

<table>
<thead>
<tr>
<th></th>
<th>Q9550</th>
<th>i7 920 (shared L3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(pthreads_barrier_wait)</td>
<td>42533</td>
<td>9820</td>
</tr>
<tr>
<td>omp barrier (icc 11.0)</td>
<td>977</td>
<td>814</td>
</tr>
<tr>
<td>gcc 4.4.3</td>
<td>41154</td>
<td>8075</td>
</tr>
</tbody>
</table>

pthreads/gcc → OS kernel call 😞

OpenMP & Intel compiler 😊

Nehalem 2 Threads

<table>
<thead>
<tr>
<th></th>
<th>Shared SMT threads</th>
<th>shared L3</th>
<th>different socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>(pthreads_barrier_wait)</td>
<td>23352</td>
<td>4796</td>
<td>49237</td>
</tr>
<tr>
<td>omp barrier (icc 11.0)</td>
<td>2761</td>
<td>479</td>
<td>1206</td>
</tr>
</tbody>
</table>

- SMT can be a performance problem for synchronizing threads
- Topology has an influence on overhead!
Thread/Process Affinity ("Pinning")

- Highly OS-dependent system calls
  - But available on all systems
    - Linux: `sched_setaffinity()`, PLPA (see below) → `hwloc`
    - 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`, `likwid-pin` (see below)
- Affinity awareness in MPI libraries
  - SGI MPT
  - OpenMPI
  - Intel MPI
  - ...
Explicit 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];
```
Process/Thread Binding With PLPA

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

```c
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:

```c
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);
}
```
How do we figure out the topology?

• … and how do we enforce the mapping without changing the code?
• Compilers and MPI libs may still give you ways to do that

• But LIKWID supports all sorts of combinations:
  
  Like
  I
  Knew
  What
  I’m
  Doing

• Open source tool collection (developed at RRZE):
  
  http://code.google.com/p/likwid
Likwid Tool Suite

• Command line tools for Linux:
  – works with standard linux 2.6 kernel
  – supports Intel and AMD CPUs
  – Supports all compilers whose OpenMP implementation is based on pthreads

• Current tools:
  – likwid-topology: Print thread and cache topology
    (similar to lstopo from the hwloc package)
  – likwid-pin: Pin threaded application without touching code
  – likwid-perfctr: Measure performance counters
  – likwid-perfscope: Performance oscilloscope w/ real-time display
  – likwid-powermeter: Current power consumption of chip (alpha stage)
  – likwid-features: View and enable/disable hardware prefetchers
  – likwid-bench: Low-level bandwidth benchmark generator tool
  – likwid-mpirun: mpirun wrapper script for easy LIKWID integration
likwid-topology – Topology information

- Based on cpuid information
- Functionality:
  - Measured clock frequency
  - Thread topology
  - Cache topology
  - Cache parameters (-c command line switch)
  - ASCII art output (-g command line switch)
- Currently supported:
  - Intel Core 2 (45nm + 65 nm)
  - Intel Nehalem, Westmere, Sandy Bridge (alpha)
  - AMD K10 (Quadcore and Hexacore)
  - AMD K8
### Output of likwid-topology

CPU name:         Intel Core i7 processor  
CPU clock:        2666683826 Hz  

**------------------------------------------------------------------**

### Hardware Thread Topology  
**------------------------------------------------------------------**

<table>
<thead>
<tr>
<th>Sockets:</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores per socket:</td>
<td>4</td>
</tr>
<tr>
<td>Threads per core:</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HWThread</th>
<th>Thread</th>
<th>Core</th>
<th>Socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

**------------------------------------------------------------------**
likwid-topology continued

Socket 0: (0 1 2 3 4 5 6 7)
Socket 1: (8 9 10 11 12 13 14 15)

*****************************************************
Cache Topology
*****************************************************
Level: 1
Size: 32 kB
Cache groups: (0 1) (2 3) (4 5) (6 7) (8 9) (10 11) (12 13) (14 15)

Level: 2
Size: 256 kB
Cache groups: (0 1) (2 3) (4 5) (6 7) (8 9) (10 11) (12 13) (14 15)

Level: 3
Size: 8 MB
Cache groups: (0 1 2 3 4 5 6 7) (8 9 10 11 12 13 14 15)

• ... and also try the ultra-cool -g option!
likwid-pin

- Inspired and based on `ptoverride` (Michael Meier, RRZE) and `taskset`
- Pins process and threads to specific cores without touching code
- Directly supports `pthread`, gcc OpenMP, Intel OpenMP
- Allows user to specify skip mask (i.e., supports many different compiler/MPI combinations)
- Can also be used as replacement for `taskset`
- Uses logical (contiguous) core numbering when running inside a restricted set of cores
- Supports logical core numbering inside node, socket, core

- Usage examples:
  - `env OMP_NUM_THREADS=6 likwid-pin -c 0,2,4-6 ./myApp parameters`
  - `env OMP_NUM_THREADS=6 likwid-pin -c S0:0-2@S1:0-2 ./myApp`
  - `env OMP_NUM_THREADS=2 mpirun -npesnode 2 \\ likwid-pin -s 0x3 -c 0,1 ./myApp parameters`
Example: STREAM benchmark on 12-core Intel Westmere: Anarchy vs. thread pinning

- No pinning
- Pinning (round-robin across sockets, physical cores first)
Topology ("mapping") choices with MPI+OpenMP:
More examples using Intel MPI+compiler & home-grown mpirun

One MPI process per node

One MPI process per socket

OpenMP threads pinned "round robin" across cores in node

Two MPI processes per socket
Case study: 3D Jacobi Solver

Basic implementation (2 arrays; no blocking etc…)

```
    do k = 1 , Nk
        do j = 1 , Nj
            do i = 1 , Ni
                y(i,j,k) = a*x(i,j,k) + b*
                (x(i-1,j,k)+ x(i+1,j,k) + x(i,j-1,k)
                +x(i,j+1,k) + x(i,j,k-1) + x(i,j,k+1))
            enddo
        enddo
    enddo
```

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**

\[ T = T_{\text{COMP}} + T_{\text{COMM}} \]

\[ T_{\text{COMP}} = \frac{N^3}{P_0} \]

\[ T_{\text{COMM}} = \frac{V_{\text{data}}}{BW} \]

\[ P_0 = 150 \text{ MLUP/s} \]

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

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

**Performance estimate** (GE) for $n$ nodes:

\[ P(n) = \frac{N^3}{\left((T_{\text{COMP}}/n) + T_{\text{COMM}}(n)\right)} \]
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 $c(\cdot) = m(\cdot, \cdot) \times b(\cdot)$
- Operation count $= 2N_{nz}$

```plaintext
do diag=1, zmax
  diagLen = jd_ptr(diag+1) - jd_ptr(diag)
  offset = jd_ptr(diag) - 1
  !$OMP PARALLEL DO
  do i=1, diagLen
    c(i) = c(i) + val(offset+i) * b(col_idx(offset+i))
  enddo
  !$OMP END PARALLEL DO
endo
```

- 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 $val[]$
- Indexed (indirect) access to RHS vector $b[]$
JDS Sparse MVM

MPI parallelization

Row-wise distribution

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 ?

• 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
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 \cdot 10^6 nonzeroes

hybrid advantage
MPI/OpenMP hybrid “how-to”: Take-home messages

- Do not use hybrid if the pure MPI code scales ok
- Be aware of intranode MPI behavior
- Always observe the topology dependence of
  - Intranode MPI
  - OpenMP overheads
- Enforce proper thread/process to core binding, using appropriate tools (whatever you use, but use SOMETHING)
- Multi-LD OpenMP processes on ccNUMA nodes require correct page placement
- Finally: Always compare the best pure MPI code with the best OpenMP code!
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
• Other options on clusters of SMP nodes
• 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 (see next section)</td>
<td>30%</td>
<td>10%</td>
</tr>
<tr>
<td>Loss by mismatch problems</td>
<td>−10%</td>
<td>−25%</td>
</tr>
<tr>
<td>Total</td>
<td>+20%</td>
<td>−15%</td>
</tr>
</tbody>
</table>

In most cases: Both categories!
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

1 x inter-socket connection per node

17 x inter-node connections per node

Sequencial ranking of MPI_COMM_WORLD

Does it matter?
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

Round robin ranking of MPI_COMM_WORLD

+ 28 x inter-node connections per node

0 x inter-socket connection per node

Never trust the default !!!
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

Two levels of domain decomposition:
- $12 \times$ inter-node connections per node
- $4 \times$ inter-socket connection per node

Bad affinity of cores to thread ranks
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

Two levels of domain decomposition

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

Good affinity of cores to thread ranks
### The Topology Problem with hybrid MPI+OpenMP

**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 Level

OpenMP

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 !!!
Numerical Optimization inside of an SMP node

2nd level of domain decomposition: OpenMP

3rd level: 2nd level cache

4th level: 1st level cache

Optimizing the numerical performance

— skipped —
The Mapping Problem with mixed model

Several multi-threaded MPI process per SMP node:

Problem
- Where are your processes and threads really located?

Solutions:
- Depends on your platform,
- e.g., with `numactl`

Further questions:
- Where is the NIC\(^1\) located?
- Which cores share caches?

---

\(^1\) NIC = Network Interface Card
Unnecessary intra-node communication

Problem:
- If several MPI process on each SMP node
  \rightarrow 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
  \rightarrow problem may be small
- MPI implementation may cause unnecessary data copying
  \rightarrow waste of memory bandwidth

Quality aspects of the MPI library
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 */
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
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


```
#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

**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

- **Master only**: MPI only outside of parallel regions
- **Multiple/only**: appl. threads inside of MPI

- **Funneled**: MPI only on master-thread
- **Multiple**: more than one thread may communicate

- **Funneled & Reserved**: reserved thread for communication
- **Funneled with Full Load Balancing**:
- **Multiple & Reserved**: reserved threads for communication
- **Multiple with Full Load Balancing**:

Different strategies to simplify the 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

Overlapping: Using OpenMP tasks

NEW OpenMP Tasking Model gives a new way to achieve more parallelism form hybrid computation.

Case study: Communication and Computation in Gyrokinetic Tokamak Simulation (GTS) shift routine

Work on particle array (packing for sending, reordering, adding after sending) can be overlapped with **data independent** MPI communication using **OpenMP tasks**.

---

Hybrid Parallel Programming

Rabenseifner, Hager, Jost

---

Slides, courtesy of Alice Koniges, NERSC, LBNL
Overlapping can be achieved with OpenMP tasks (1\textsuperscript{st} part)

Overlapping MPI\_Allreduce with particle work

- **Overlap**: Master thread encounters !$omp master\) tasking statements and creates work for the thread team for deferred execution. MPI Allreduce call is immediately executed.
- MPI implementation has to support at least MPI\_THREAD\_FUNNELED
- Subdividing tasks into smaller chunks to allow better *load balancing* and *scalability* among threads.

Slides, courtesy of Alice Koniges, NERSC, LBNL
Overlapping can be achieved with OpenMP tasks (2\textsuperscript{nd} part)

```
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!}
```

Overlapping particle reordering

Particle reordering of remaining particles (above) and adding sent particles into array (right) & sending or receiving of shifted particles can be independently executed.
OpenMP tasking version outperforms original shifter, especially in larger poloidal domains

- Performance breakdown of GTS shifter routine using 4 OpenMP threads per MPI process with varying domain decomposition and particles per cell on Franklin Cray XT4.
- MPI communication in the shift phase uses a toroidal MPI communicator (constantly 128).
- Large performance differences in the 256 MPI run compared to 2048 MPI run!
- Speed-Up is expected to be higher on larger GTS runs with hundreds of thousands CPUs since MPI communication is more expensive.
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
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

Terrible with non-default schedule

No speedup with 1000x1000
Second CPU only usable in small cases
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 Masteronly style:
  75% of the threads sleep while master thread communicates

• With Overlapping Comm. & Comp.:
  Master thread should be only partially reserved for communication
  – otherwise too expensive

• MPI library must support
  – Multiple threads
  – Two fabrics (shmem + 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
• Other options on clusters of SMP nodes
• 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

```c
#pragma omp parallel for schedule(dynamic)
for (i=0; i<n; i++) {
    /* poorly balanced iterations */ ...
}
```
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
Using more OpenMP threads could reduce the memory usage **substantially**, up to **five** times on Hopper Cray XT5 (eight-core nodes).

**Case study: MPI+OpenMP memory usage of NPB**

**Graph:**
- **XT5:**
  - 256*1: BT-MZ 1.2, SP-MZ 0.4
  - 128*2: BT-MZ 0.8, SP-MZ 0.6
  - 64*4: BT-MZ 0.4, SP-MZ 0.2
  - 32*8: BT-MZ 0.2, SP-MZ 0.1

**Legend:**
- BT-MZ
- SP-MZ

**Note:** Always same number of cores

**Citation:**
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 threads per MPI process?

- 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
    \rightarrow overlapping of MPI and computation may be necessary,
    \rightarrow some NICs unused?
  - Too few threads
    \rightarrow too much memory consumption (see previous slides)
- Optimum
  - somewhere between 1 and \(m \times n\) threads per MPI process,
  - Typically:
    - \textbf{Optimum} = \textbf{n}, i.e., 1 MPI process per socket
    - \textbf{Sometimes} = \textbf{n/2}, i.e., 2 MPI processes per socket
    - \textbf{Seldom} = \textbf{2n}, i.e., each MPI process on 2 sockets
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 MPI 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

- 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
- Other options on clusters of SMP nodes
- Summary
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 * thread_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!

  ```
  !$OMP BARRIER
  !$OMP MASTER
  call MPI_Xxx(...)  #pragma omp master
  !$OMP END MASTER
  !$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
!$OMP PARALLEL
!$OMP DO
   do i=1,1000
      a(i) = buf(i)
   end do
!$OMP END DO NOWAIT
!$OMP BARRIER
!$OMP MASTER
   call MPI_RECV(buf,...)
!$OMP END MASTER
!$OMP BARRIER
!$OMP DO
   do i=1,1000
      c(i) = buf(i)
   end do
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

```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>MPIch2-1.1.0a2</td>
<td>ch:nemesis has “Initial Thread-support”</td>
</tr>
<tr>
<td>Intel MPI 3.1</td>
<td>ch3:nemesis (default) has MPI_THREAD_MULTIPLE</td>
</tr>
<tr>
<td>SciCortex MPI</td>
<td>Full MPI_THREAD_MULTIPLE</td>
</tr>
<tr>
<td>HP MPI-2.2.7</td>
<td>MPI_THREAD_FUNNELED</td>
</tr>
<tr>
<td>SGI MPT-1.14</td>
<td>Full MPI_THREAD_MULTIPLE (with libmtmpi)</td>
</tr>
<tr>
<td>IBM MPI</td>
<td>Not thread-safe?</td>
</tr>
<tr>
<td>Nec MPI/SX</td>
<td>Full MPI_THREAD_MULTIPLE</td>
</tr>
<tr>
<td></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**

• Other options on clusters of SMP nodes
• Summary
Thread Correctness – Intel ThreadChecker  1/3

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

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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>1</td>
<td>Write -&gt; Error</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 1  | Write da          |          |         |             |         |         |
|    | ad_race "pthread_race.c":31 conflicts with |         |         |             |         |         |
|    | d_race.d_race.c" |          |         |             |         |         |
| 1  | ta-race           |          |         |             |         |         |
|    | e.c":2 a prior memory write of |         |         |             |         |         |
|    | c":31             |          |         |             |         |         |

| 1  |                  |          |         |             |         |         |
|    |                  |          |         |             |         |         |

| 1  |                  |          |         |             |         |         |
|    |                  |          |         |             |         |         |

With new Intel Inspector XE 2011: Command line interface must be used within mpirun / mpiexec.
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_url)
Thread Correctness – Intel ThreadChecker  3/3

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

```bash
configure --enable-mpi-threads
    --enable-debug
    --enable-mca-no-build=memory-ptmalloc2
CC=icc F77=ifort FC=ifort
CFLAGS=’-debug all -inline-debug-info tcheck’
CXXFLAGS=’-debug all -inline-debug-info tcheck’
FFLAGS=’-debug all -tcheck’
LDFLAGS=’tcheck’
```

• Then run with:

```bash
mpirun --mca tcp,sm,self -np 2 tcheck_cl
    --reinstrument -u full --format html
    --cache_dir '/tmp/my_username_${__tc_cl_cache}'
    --report 'tc_mpi_test_suite_${}'
    --options 'file=tc_my_executable_%H_%I,'
    pad=128, delay=2, stall=2'
    ./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
Scalasca – Example “Wait at Barrier”

Indication of non-optimal load balance
Scalasca – 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

• Other options on clusters of SMP nodes
  – Pure MPI – multi-core aware (Rolf Rabenseifner)
  – Remarks on MPI scalability / Cache Optimization / Cost-benefit /PGAS (R.R.)
  – Hybrid programming and accelerators (Gabriele Jost)

• Summary
Pure MPI – multi-core aware

- Hierarchical domain decomposition (or distribution of Cartesian arrays)

Domain decomposition:
1 sub-domain / SMP node

Further partitioning:
1 sub-domain / socket

1 / core

Cache optimization:
Blocking inside of each core, block size relates to cache size. 1-3 cache levels.

Example on 10 nodes, each with 4 sockets, each with 6 cores.
How to achieve a hierarchical domain decomposition (DD)?

- **Cartesian grids:**
  - Several levels of subdivide
  - Ranking of MPI_COMM_WORLD – three choices:
    a) Sequential ranks through original data structure + locating these ranks correctly on the hardware
       - can be achieved with one-level DD on finest grid + special startup (mpiexec) with optimized rank-mapping
    b) Sequential ranks in comm_cart (from MPI_CART_CREATE)
       - requires optimized MPI_CART_CREATE, or special startup (mpiexec) with optimized rank-mapping
    c) Sequential ranks in MPI_COMM_WORLD + additional communicator with sequential ranks in the data structure + self-written and optimized rank mapping.

- **Unstructured grids:**
  → next slide
How to achieve a hierarchical domain decomposition (DD)?

- Unstructured grids:
  - Multi-level DD:
    - Top-down: Several levels of (Par)Metis
      - unbalanced communication
      - demonstrated on next (skipped) slide
    - Bottom-up: Low level DD
      + higher level recombination
      - based on DD of the grid of subdomains
  - Single-level DD (finest level)
    - Analysis of the communication pattern in a first run (with only a few iterations)
    - Optimized rank mapping to the hardware before production run
    - E.g., with CrayPAT + CrayApprentice
Top-down – several levels of (Par)Metis

Steps:
- Load-balancing (e.g., with ParMetis) on outer level, i.e., between all SMP nodes
- Independent (Par)Metis inside of each node
- Metis inside of each socket
  - Subdivide does not care on balancing of the outer boundary
  - processes can get a lot of neighbors with inter-node communication
  - unbalanced communication
Bottom-up – Multi-level DD through recombination

1. Core-level DD: partitioning of application’s data grid
2. Numa-domain-level DD: recombining of core-domains
3. SMP node level DD: recombining of socket-domains

- **Problem:**
  Recombination must **not** calculate patches that are smaller or larger than the average

- **Advantage:**
  Communication is balanced!

Graph of all sub-domains (core-sized)

Divided into sub-graphs for each socket
Profiling solution

- First run with profiling
  - Analysis of the communication pattern
- Optimization step
  - Calculation of an optimal mapping of ranks in MPI_COMM_WORLD to the hardware grid (physical cores / sockets / SMP nodes)
- Restart of the application with this optimized locating of the ranks on the hardware grid

- Example: CrayPat and CrayApprentice
Scalability of MPI to hundreds of thousands ...

Scalability of pure MPI

- As long as the application does not use
  - MPI_ALLTOALL
  - MPI_<collectives>V (i.e., with length arrays)
and application
  - distributes all data arrays
one can expect:
  - Significant, but still scalable memory overhead for halo cells.
  - MPI library is internally scalable:
    - **E.g., mapping ranks \( \rightarrow \) hardware grid**
      - Centralized storing in shared memory (OS level)
      - In each MPI process, only used neighbor ranks are stored (cached) in process-local memory.
    - **Tree based algorithm with \( O(\log N) \)**
      - From 1000 to 1000,000 process \( O(\log N) \) only doubles!
Remarks on Cache Optimization

• **After** all parallelization domain decompositions (DD, up to 3 levels) are done:
  • Cache-blocking is an additional DD into data blocks
    – that fit to 2nd or 3rd level cache.
    – It is done inside of each MPI process (on each core).
    – Outer loops run from block to block
    – Inner loops inside of each block
    – Cartesian example: 3-dim loop is split into
      
      ```
      do i_block=1,ni,stride_i
          do j_block=1,nj,stride_j
              do k_block=1,nk,stride_k
                  do i=i_block,min(i_block+stride_i-1, ni)
                      do j=j_block,min(j_block+stride_j-1, nj)
                          do k=k_block,min(k_block+stride_k-1, nk)
                              a(i,j,k) = f( b(i±0,1,2, j±0,1,2, k±0,1,2) )
                          end do
                      end do
                  end do
              end do
          end do
      end do
      
      Access to 13-point stencil
      ```
Remarks on Cost-Benefit Calculation

Costs
- for optimization effort
  - e.g., additional OpenMP parallelization
  - e.g., 3 person month $\times 5,000 \text{ €} = 15,000 \text{ €}$ (full costs)

Benefit
- from reduced CPU utilization
  - e.g., Example 1:
    100,000 € **hardware costs** of the cluster
    $\times 20\%$ used by this application over whole lifetime of the cluster
    $\times 7\%$ performance win through the optimization
    $= 1,400 \text{ €} \rightarrow \text{total loss} = 13,600 \text{ €}$
  - e.g., Example 2:
    10 Mio € **system** $\times 5\%$ used $\times 8\%$ performance win
    $= 40,000 \text{ €} \rightarrow \text{total win} = 25,000 \text{ €}$
Remarks on MPI and PGAS (UPC & CAF)

- Parallelization always means
  - expressing locality.

- If the application has no locality,
  - Then the parallelization needs not to model locality
    \( \rightarrow \) UPC with its round robin data distribution may fit

- If the application has locality,
  - then it must be expressed in the parallelization

- Coarray Fortran (CAF) expresses data locality explicitly through “co-dimension”:
  - \( A(17,15)[3] \)
  - = element \( A(17,13) \) in the distributed array \( A \) in process with rank 3
Remarks on MPI and PGAS (UPC & CAF)

- Future shrinking of memory per core implies
  - Communication time becomes a bottleneck
  - Computation and communication must be overlapped, i.e., latency hiding is needed

- With PGAS, halos are not needed.
  - But it is hard for the compiler to access data such early that the transfer can be overlapped with enough computation.

- With MPI, typically too large message chunks are transferred.
  - This problem also complicates overlapping.

- Strided transfer is expected to be slower than contiguous transfers
  - Typical packing strategies do not work for PGAS on compiler level
  - Only with MPI, or with explicit application programming with PGAS
Remarks on MPI and PGAS (UPC & CAF)

• Point-to-point neighbor communication
  – PGAS or MPI nonblocking may fit if message size makes sense for overlapping.

• Collective communication
  – Library routines are best optimized
  – Non-blocking collectives (comes with MPI-3.0) versus calling MPI from additional communication thread
  – Only blocking collectives in PGAS library?
Remarks on MPI and PGAS (UPC & CAF)

• For extreme HPC (many nodes x many cores)
  – Most parallelization may still use MPI
  – Parts are optimized with PGAS, e.g., for better latency hiding
  – PGAS efficiency is less portable than MPI
  – #ifdef … PGAS
  – Requires mixed programming PGAS & MPI
  → will be addressed by MPI-3.0
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

**Other options on clusters of SMP nodes**

- Pure MPI – multi-core aware  (Rolf Rabenseifner)
- Remarks on MPI scalability / Cache Optimization / Cost-benefit /PGAS (R.R.)
- Hybrid programming and accelerators  (Gabriele Jost)

- Summary
Hybrid Programming and Accelerators

- Under Discussion OpenMP support for Accelerators in 4.0
  - To be announced at SC12
  - Multiple devices of the same type (homogeneous)
  - Device type known at compile time
  - Automatic run-time and programmed user-control device selection
  - Structured and unstructured block data placement
    - Data regions and mirror directives
  - Synchronous and asynchronous data-movement
  - Accelerator style parallel launch with multiple 'threads' of execution on the device: eg accelerator parallel regions
  - Dispatch-style parallel launch (offload) to a single thread of execution on the device; eg accelerator tasks
OpenMP Accelerator Memory Model

- **Current Memory Model:**
  - Relaxed-Consistency Shared-Memory
  - All threads have access to the memory
  - Data-sharing attributes: shared, private

- **Proposed Additions to Memory Model**
  - Separate Host and Accelerator Memory
  - Data Movement Host<-Accelerator indicated by compiler directives
  - Updates to different memories indicated by compiler directives
  - `#pragma omp acc_data [clause]`
    - `acc_shared`
    - `acc_copyout, acc_copyin`
OpenMP Accelerator Execution Model

- Current OpenMP Execution Model:
  - Execution starts single threaded
  - Fork-Join Threads at OpenMP parallel regions
  - Work-sharing indicated via compiler directives

- Proposed additions to the Execution Model:
  - Explicit accelerator regions or tasks are generated at beginning of accelerator regions

  - \#pragma acc_region [clause]
    - Purpose: Define code that is to be run on accelerator
    - acc_copyin (list)
    - acc_copyout (list)
  
  \#pragma omp acc_loop [clause]
Other Compiler Directive Based APIs

• **OpenACC:**
  - Support of separate host and device memory: copy-in, copy-out, etc.
  - Support to execute compute kernels on the accelerator device
  - Fine grained control of execution on accelerator: num_gangs, num_workers, vector length, etc

• **PGI Compiler Directives:**
  - Similar to OpenMP (see example)

• **Compiler Directives for Many Core Architectures:**
  - Generate tasks for parts of the code to be off-loaded to many core nodes
Example: Jacobi Iteration OpenMP directives

```c
!$OMP PARALLEL DO PRIVATE(i,j,k)
    DO k = 1, Z, 1
        DO j = 1, Y, 1
            DO i = 1, X, 1
                data(i,j,k,new) = &
                ( data(i,j,k,old) + &
                  data(i,j-1,k,old) + data(i,j+1,k,old) + &
                  data(i,j,k-1,old) + data2(i,j,k+1,old) - &
                  edge(i,j,k) ) / 6.0
            END DO
        END DO
    END DO
END DO
END DO
```
Test Case: Hybrid Jacobi using PGI directives

- PGI (http://www.pgroup.com) provides compiler directives for accelerators
  - Website for some documentation
- PGI active member of OpenMP Language committee
  - Use PGI Directives
- OpenMP Language committee at this time closely follows path set by PGI
- Original Hybrid MPI/OpenMP implementation provided by courtesy of EPCC (Edinburgh Parallel Computing Center) (http://www.epcc.ed.ac.uk)
Test System

- TACC's Dell XD Visualization Cluster Longhorn (http://www.tacc.utexas.edu/user-services/user-guides/longhorn-user-guide)
- 240 nodes containing 48GB of RAM,
- 8 Intel Nehalem cores (@ 2.5 GHz), and 2 NVIDIA Quadro FX 5800 GPUs per node
- Test System: Longhorn at TACC
- pgf90 11.5
- -fastsse -ta=nvidia,time -Minfo=vect,accel -Mcuda=cuda3.2
Unoptimized

```fortran
!$omp acc_region
DO k = 1, Z, 1
  DO j = 1, Y, 1
    DO i = 1, X, 1
      data(i,j,k,new) = &
      ( data(i,j,k,old) + &
        data(i,j-1,k,old) + &
        data(i,j+1,k,old) + &
        data(i,j,k-1,old) + &
        data2(i,j,k+1,old) - &
        edge(i,j,k) ) / 6.0
    END DO
  END DO
END DO
!$omp end acc_region
```

No performance increase when using accelerator

jacobistep:
59, Loop carried dependence of 'data' prevents parallelization
   Loop carried backward dependence of 'data' prevents vectorization
60, Loop carried dependence of 'data' prevents parallelization
   Loop carried backward dependence of 'data' prevents vectorization
61, Loop carried dependence of 'data' prevents parallelization
   Loop carried backward dependence of 'data' prevents vectorization

Accelerator kernel generated
59, !$acc do seq
60, !$acc do seq
61, !$acc do seq

Non-stride-1 accesses for array 'data'
Non-stride-1 accesses for array 'edge'
Optimized for parallelization:

```c
!$acc data region local(temp2)
updatein(data(0:X+1,0:Y+1,0:Z+1,old))
updateout(data(0:X+1,0:Y+1,0:Z+1,new)) updatein(edge)
!$acc region
  temp2 = data (:,:,:,:old)
  DO k = 1, Z, 1
    DO j = 1, Y, 1
      DO i = 1, X, 1
        data(i,j,k,new) =
          temp2(i-1,j,k) + temp2(i+1,j,k) +
          ... 
          edge(i,j,k) / 6.0
      END DO
    END DO
  END DO
!$acc end region
!$acc end data region
```

- copy to temporary array to expose non-overlap

244, Loop is parallelizable
245, Loop is parallelizable
246, Loop is parallelizable

Accelerator kernel generated
244, !$acc do parallel, vector(4) ! blockidx%y threadidx%z
245, !$acc do parallel, vector(4) ! blockidx%x threadidx%y
246, !$acc do vector(16) ! threadidx%x

Cached references to size [18x6x6] block of 'temp2'
Optimized for parallelization alternative: Compiler directives

```fortran
!$acc data region local(temp2)
updatein(data(0:X+1,0:Y+1,0:Z+1,old))
updateout(data(0:X+1,0:Y+1,0:Z+1,new)) updatein(edge)
!$acc region
!$acc do parallel,independent
  DO k = 1, Z, 1
!$acc do parallel,independent
  DO j = 1, Y, 1
!$acc do parallel independent
  DO i = 1, X, 1
    data(i,j,k,new) = &
    ( data(i-1,j,k,old) + data(i+1,j,k,old) + &
      data(i,j-1,k,old) + data(i,j+1,k,old) + &
      data(i,j,k-1,old) + data(i,j,k+1,old) - &
      edge(i,j,k) ) / 6.0
  END DO
END DO
END DO
!$acc end region
!$acc end data region
```

Indicate non-overlap using “independent” clause
Optimized for parallelization alternative: Compiler directives

```c
!$acc data region local(temp2)
updatein(data(0:0,X+1,0:Y+1,0:Z+1,old))
updateout(data(0:0,X+1,0:Y+1,0:Z+1,new))
updatein(edge)

!$acc region !$acc do parallel,independent
DO k = 1, Z, 1
  !$acc do parallel independent
  DO j = 1, Y, 1
    !$acc do parallel independent
    DO i = 1, X, 1
      data(i,j,k,new) = 
      ( data(i-1,j,k,old) + data(i+1,j,k,old) + 
        data(i,j-1,k,old) + data(i,j+1,k,old) + 
        data(i,j,k-1,old) + data(i,j,k+1,old) - 
        edge(i,j,k) ) / 6.0
    END DO
  END DO
END DO

!$acc end region !$acc end data region
```

Accelerator kernel generated

238, !$acc do parallel, vector(2) ! blockidx%y threadidx%z
240, !$acc do parallel, vector(8) ! blockidx%x threadidx%y
242, !$acc do vector(8) ! threadidx%x
   Non-stride-1 accesses for array 'data'
252, Generating !$acc update host(data(0:x+1,0:y+1,0:z+1,:new))
255, Generating !$acc update device(data(0:x+1,0:y+1,0:z+1,:old))
259, Loop is parallelizable
261, Loop is parallelizable
263, Loop is parallelizable
   Accelerator kernel generated
259, !$acc do parallel, vector(2) ! blockidx%y threadidx%z
261, !$acc do parallel, vector(8) ! blockidx%x threadidx%y
263, !$acc do vector(8) ! threadidx%x
Using different Devices

```fortran
!$acc data region local(temp2)
updatein(data(0:X+1,0:Y+1,0:Z+1,old))
updateout(data(0:X+1,0:Y+1,0:Z+1,new)) updatein(edge)

!$acc region
  temp2 = data (:,:,:old)
  DO k = 1, Z, 1
    DO j = 1, Y, 1
      DO i = 1, X, 1
        data(i,j,k,new) = ( temp2(i-1,j,k) + temp2(i+1,j,k) + ……
        & temp2(i-1,j,k) + temp2(i+1,j,k) + edge(i,j,k) ) / 6.0
      END DO
    END DO
  END DO
!$acc end region
if (first) then
  macc = MOD(rank,2)+1
  call acc_set_device_num(macc,acc_device_type)
endif

Use different devices for different MPI processes
```
module glob
    real (kind(1.0e0)), dimension(:,:,:,:), allocatable,pinned :: data
    real (kind(1.0e0)), dimension(:,:,:,:), allocatable,pinned :: edge
    logical first
!$acc mirror(data,edge)
end module glob

!$acc data region local(temp2)
updatein(data(0:X+1,0:Y+1,0:Z+1,old))
updateout(data(0:X+1,0:Y+1,0:Z+1,new)) updatein(edge)
!$acc region
    temp2 = data (:,:,:old)
    DO k = 1, Z, 1
        DO j = 1, Y, 1
            DO i = 1, X, 1
                data(i,j,k,new) = ( temp2(i-1,j,k) + temp2(i+1,j,k) + ... edge (I,j,k))/6.
            END DO
        END DO
    END DO
!$acc end region
!$acc end data region
Compare PGI, openACC and OpenMP extensions

module glob
    real (kind(1.0e0)), dimension(:,:,,:), allocatable, pinned :: data
    real (kind(1.0e0)), dimension(:,:,,:), allocatable, pinned :: edge
    logical first
!$acc mirror(data,edge)
end module glob

 !$acc data region local(temp2) OpenMP and OpenACC
 updatein(data(0:X+1,0:Y+1,0:Z+1,old))
 updateout(data(0:X+1,0:Y+1,0:Z+1,new)) updatein(edge)
 !$acc region OpenMP: planned, OpenACC: copyin clause
 temp2 = data(:,:,old)
 DO k = 1, Z, 1
   DO j = 1, Y, 1
     DO i = 1, X, 1
       data(i,j,k,new) = ( temp2(i-1,j,k) + temp2(i+1,j,k) + ... edge(I,j,k))/6.
     END DO
   END DO
 END DO
 !$acc end region OpenMP and OpenACC
 !$acc end data region

OpenACC provides more detailed control of how the kernel should be executed

Concluding Remarks

Still many open questions:
• Multi-core vs accelerator: General purpose vs specialized, e.g.:
  - GPU runs kernels independently
  - GPU accelerator has large team of threads
  - GPU thread counts exceed number of cores
  - GPU uses scheduling algorithm to hide memory latency, synchronize threads into groups.
  - Stream processing
• How do we address parallelism within accelerator?
• Other types of co-processors?
• Which of the differences should be exposed via OpenMP?
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
  - Other options on clusters of SMP nodes

- Summary
Acknowledgements

- We want to thank
  - Gerhard Wellein, RRZE
  - Alice Koniges, NERSC, LBNL
  - Rainer Keller, HLRS and ORNL
  - Jim Cownie, Intel
  - SCALASCA/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 – hybrid MPI+OpenMP

MPI + OpenMP

- Seen with NPB-MZ examples
  - BT-MZ → strong improvement (as expected)
  - SP-MZ → small improvement
  - Usability on higher number of cores
- Advantages
  - Memory consumption
  - Load balancing
  - 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”
- Huge amount of pitfalls
- Optimum: Somewhere in the area of 1 MPI process per NUMA domain
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 worksharing support → using OpenMP tasks as workaround
Summary – good and bad

- Optimization
  - 1 MPI process per core …………………………………… ..… per SMP node
  - mismatch problem
  ^– 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 → GPU, FPGA, …
  – 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: 20% Introductory, 50% Intermediate, 30% Advanced)

Authors.  Rolf Rabenseifner, HLRS, University of Stuttgart, Germany
Georg Hager, University of Erlangen-Nuremberg, Germany
Gabriele Jost, Supersmith, Maximum Performance Software, USA

Abstract. Most HPC systems are clusters of shared memory nodes. Such systems can be PC clusters with single/multi-socket and multi-core SMP nodes, but also “constellation” type systems with large SMP nodes. Parallel programming may combine the distributed memory parallelization on the node interconnect with the shared memory parallelization inside of each node.

This tutorial analyzes the strengths and weaknesses of several parallel programming models on clusters of SMP nodes. Multi-socket-multi-core systems in highly parallel environments are given special consideration. This includes a discussion on planned future OpenMP support for accelerators. Various hybrid MPI+OpenMP approaches are compared with pure MPI, and benchmark results on different platforms are presented. Numerous case studies demonstrate the performance-related aspects of hybrid MPI/OpenMP programming, and application categories that can take advantage of this model are identified. Tools for hybrid programming such as thread/process placement support and performance analysis are presented in a "how-to" section.

Rolf Rabenseifner

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, and in Jan. 2012, he was appointed as GCS' PATC director.
Georg Hager holds a PhD in computational physics from the University of Greifswald. He has been working with high performance systems since 1995, and is now a senior research scientist in the HPC group at Erlangen Regional Computing Center (RRZE). His daily work encompasses all aspects of HPC user support and training, assessment of novel system and processor architectures, and supervision of student projects and theses. Recent research includes architecture-specific optimization for current microprocessors, performance modeling on processor and system levels, and the efficient use of hybrid parallel systems. His textbook “Introduction to High Performance Computing for Scientists and Engineers” is recommended reading for many HPC-related courses and lectures worldwide. A full list of publications, talks, and other things he is interested in can be found in his blog: http://blogs.fau.de/hager.
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 2005 she moved from California to the Pacific Northwest and joined Sun Microsystems as a staff engineer in the Compiler Performance Engineering team, analyzing compiler generated code and providing feedback and suggestions for improvement to the compiler group. She then decided to explore the world beyond scientific computing and joined Oracle as a Principal Engineer working on performance analysis for application server software. That was fun, but she realized that her real passions remain in area of performance analysis and evaluation of programming paradigms for high performance computing and joined the Texas Advanced Computing Center (TACC), working on all sorts of exciting projects related to large scale parallel processing for scientific computing. In 2011, she joined Advanced Micro Devices (AMD) as a design engineer in the Systems Performance Optimization group.
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, 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.


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

• Georg Hager and Gerhard Wellein: 
  *Introduction to High Performance Computing for Scientists and Engineers.*

• Barbara Chapman et al.: 
  *Toward Enhancing OpenMP’s Work-Sharing Directives.*

• Barbara Chapman, Gabriele Jost, and Ruud van der Pas: 
  *Using OpenMP.*

• Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur and Jesper Larsson Traeff: 
  *MPI on a Million Processors.*
  EuroPVM/MPI 2009, Springer.

• Alice Koniges et al.: *Application Acceleration on Current and Future Cray Platforms.*

References

• J. Treibig, G. Hager and G. Wellein:  
  **LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments.**  

• H. Stengel:  
  **Parallel programming on hybrid hardware: Models and applications.**  
  http://www.hpc.rrze.uni-erlangen.de/Projekte/hybrid.shtml
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


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 21
  – Dell Linux Cluster Lonestar 22
  – NUMA Control (numactl) 24
  – On Cray XE6 Hermit (AMD Interlagos) 33
  – On Cray XE6 Hector (AMD Magny Cours) 37
  – On a IBM Power6 system 45
  – Conclusions 48

• Practical “How-To” on hybrid programming ....... 50
  – How to compile, link and run 52
  – Running the code efficiently? 58
  – A short introduction to ccNUMA 60
  – ccNUMA Memory Locality Problems / First Touch 62
  – ccNUMA problems beyond first touch 66
  – Bandwidth and latency 68

  – Parallel vector triad benchmark 71
  – OpenMP overhead 78
  – Thread/Process Affinity (“Pinning”) 79
  – LIKWID 82
  – Example: 3D Jacobi Solver 90
  – Example: Sparse Matrix-Vector-Multiply with JDS 93
  – Hybrid MPI/OpenMP “how-to”: Take-home mess. 99

• Mismatch Problems ................................. 100
  – Topology problem 102
  – Mapping problem with mixed model 109
  – Unnecessary intra-node communication 110
  – Sleeping threads and network saturation problem 111
  – Additional OpenMP overhead 112
  – Overlapping communication and computation 113
  – Communication overhead with DSM 122
  – Back to the mixed model 127
  – No silver bullet 128

• Opportunities: Application categories that can benefit from hybrid parallelization 129
  – Nested Parallelism 130
  – Load-Balancing 131
  – Memory consumption 132
Content

- Opportunities, if MPI speedup is limited due to algorithmic problem 136
- To overcome MPI scaling problems 137
- Summary 138

• Thread-safety quality of MPI libraries . . . . . . . . . . . . . . . . . . 139
  - MPI rules with OpenMP 140
  - Thread support of MPI libraries 143
  - Thread Support within OpenMPI 144

• Tools for debugging and profiling MPI+OpenMP . . 145
  - Intel ThreadChecker 146
  - Performance Tools Support for Hybrid Code 148

• Other options on clusters of SMP nodes . . . . . . . . 152
  - Pure MPI – multi-core aware 153
  - Hierarchical domain decomposition 154
  - Scalability of MPI to hundreds of thousands 159
  - Remarks on Cache Optimization 160
  - Remarks on Cost-Benefit Calculation 161
  - Remarks on MPI and PGAS (UPC & CAF) 162
  - Hybrid Programming and Accelerators 166

• Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 182
  - Acknowledgements 183
  - Summaries 184
  - Conclusions 189

• Appendix . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 190
  - Abstract 191
  - Authors 192
  - References (with direct relation to the content of this tutorial) 195
  - Further references 199

• Content . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 206