Hybrid MPI & OpenMP Parallel Programming

MPI + OpenMP and other models on clusters of SMP nodes

Rolf Rabenseifner\(^1\)  Georg Hager\(^2\)  Gabriele Jost\(^3\)

Rabenseifner@hlrs.de  Georg.Hager@rrze.uni-erlangen.de  gjost@tacc.utexas.edu

\(^1\) High Performance Computing Center (HLRS), University of Stuttgart, Germany
\(^2\) Regional Computing Center (RRZE), University of Erlangen, Germany
\(^3\) Texas Advanced Computing Center, The University of Texas at Austin, USA

PRACE Summer School: “Taking the Most Out of Supercomputers”
Aug. 29 – Sep. 1, 2011, Espoo, CSC, Finland

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

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

- 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?

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
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!

Hybrid Masteronly

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

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

Overlap Communication and Computation

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!
Outline

- Introduction / Motivation
- Programming models on clusters of SMP nodes
- Case Studies / pure MPI vs hybrid MPI+OpenMP
  - The Multi-Zone NAS Parallel Benchmarks
  - For each application we discuss:
    - Benchmark implementations based on different strategies and programming paradigms
    - Performance results and analysis on different hardware architectures
  - Compilation and Execution Summary

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

- Practical “How-To” on hybrid programming
- Mismatch Problems
- Opportunities: Application categories that can benefit from hybrid parallelism.
- Thread-safety quality of MPI libraries
- Tools for debugging and profiling MPI+OpenMP
- Other options on clusters of SMP nodes
- Summary

The Multi-Zone NAS Parallel Benchmarks

<table>
<thead>
<tr>
<th>Time step</th>
<th>sequential</th>
<th>sequential</th>
<th>sequential</th>
</tr>
</thead>
<tbody>
<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>

- 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

Using MPI/OpenMP: ADI Method

call omp_set_numthreads (weight)
do step = 1, itmax
call exch_qbc(u, qbc, nx, ...)
call mpi_send/recv
subroutine zsolve(u, rsd, ...)
...
!
subroutine ssor
!
subroutine sinc1(
do k = 2, nz-1
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP& PRIVATE(m,i,j,k...)!

$OMP DO
do j = 2, ny-1
!$OMP DO
do i = 2, nx-1
!$OMP DO
do m = 1, 5
!
rsd(m,i,j,k)=
dr*sin(m,i,j,k-1)
end do
end do
end do
end do

$OMP END PARALLEL

Pipelined Thread Execution in SSOR

subroutine sinc1
!
subroutine sinc2
!
subroutine sinc3

!$OMP PARALLEL DEFAULT(SHARED)
!$OMP4 PRIVATE(m,i,j,k...)
!$OMP DO
do k = 2, nz-1
!$OMP DO
do j = 2, ny-1
!$OMP DO
do i = 2, nx-1
!$OMP DO
do m = 1, 5
!
rsd(m,i,j,k)=
dr*sin(m,i,j,k-1)
end do
end do
end do
end do

!$OMP END PARALLEL
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

Benchmark Architectures

- Sun Constellation (Ranger)
- Cray XT5 (skipped)
- IBM Power 6

Sun Constellation Cluster Ranger (1)

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

Sun Constellation Cluster Ranger (2)

- Compilation:
  - PGI pgf90 7.1
  - mptf90 -tp barceloana-64 -r8 -mp

- Cache optimized benchmarks Execution:
  - MPI MVAPICH
  - setenv OMP_NUM_THREADS nthreads
  - btrun numaclt bt-mz.exe

- numaclt controls
  - Socket affinity: select sockets to run
  - Core affinity: select cores within socket
  - Memory policy: where to allocate memory
SUN: Running hybrid on Sun Constellation
Cluster Ranger

- Highly hierarchical
- Shared Memory:
  - Cache-coherent, Non-uniform memory access (ccNUMA) 16-way Node (Blade)
- Distributed memory:
  - Network of ccNUMA blades
    - Core-to-Core
    - Socket-to-Socket
    - Blade-to-Blade
    - Chassis-to-Chassis

SUN: NPB-MZ Class E Scalability on Ranger

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

NUMA Control: Process Placement

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

**Command:**
```
numactl <options> ./a.out
```

**Example:**
```
umactl -N 1 ./a.out
```
```
umactl -c 0,1 ./a.out
```

NUMA Operations: Memory Placement

- Memory allocation:
  - MPI
    - local allocation is best
  - OpenMP
    - Interleave best for large, completely shared arrays that are randomly accessed by different threads
    - local best for private arrays
  - Once allocated, a memory-structure is fixed

**Example:**
```
numactl -N -1 -1 ./a.out
```
**NUMA Operations (cont. 3)**

<table>
<thead>
<tr>
<th>cmd</th>
<th>option</th>
<th>arguments</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numactl</td>
<td>-N</td>
<td>[0,1,2,3]</td>
<td>Only execute process on cores of this (these) socket(s).</td>
</tr>
<tr>
<td>numactl</td>
<td>-l</td>
<td>[0,1,2,3]</td>
<td>Allocate round robin (interleave) on these sockets.</td>
</tr>
<tr>
<td>numactl</td>
<td>-l</td>
<td>[0,1,2,3]</td>
<td>Allocate on current socket.</td>
</tr>
<tr>
<td>numactl</td>
<td>-p</td>
<td>[0,1,2,3]</td>
<td>Select only one.</td>
</tr>
<tr>
<td>numactl</td>
<td>-n</td>
<td>[0,1,2,3]</td>
<td>Only allocate on this (these) socket(s).</td>
</tr>
<tr>
<td>numactl</td>
<td>-C</td>
<td>[0,1,2,3, 4,5,6,7, 8,9,10,11, 12,13,14,15,16]</td>
<td>Only execute process on this (these) Core(s).</td>
</tr>
</tbody>
</table>

**Numactl – Pitfalls:**

**Using Threads across Sockets**

- `bl-mz.1024x8` yields best load balance

  - `--pe 2way 8192`
  - `export OMP_NUM_THREADS=8`
  - `my_rank=$PMI_RANK`
  - `local_rank=$(( $my_rank % $myway ))`
  - `numnode=$(( $local_rank + 1 ))`

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

  **Bad performance!**
  - Each process runs 8 threads on 4 cores
  - Memory allocated on one socket

**Hybrid Batch Script: 4 tasks, 4 threads/task**

- **job script (Rouane shell):**

  ```
  #!-pe 4way 32
  export OMP_NUM_THREADS=4
  ibrun numa.sh
  ```

- **job script (C shell):**

  ```
  #!-pe 4way 32
  setenv OMP_NUM_THREADS 4
  ibrun numa.csh
  ```

**Numactl – Pitfalls:**

**Using Threads across Sockets**

- `bl-mz.1024x8`
  - `export OMP_NUM_THREADS=8`
  - `num_node=$PMI_RANK`
  - `local_rank=$(( $num_rank % $numway ))`
  - `numnode=$(( $local_rank + 1 ))`

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

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

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

- Results obtained by the courtesy of the HPCMO Program and the Engineer Research and Development Center Major Shared Resource Center, Vicksburg, MS (http://www.erdc.hpc.mil/index)
- Cray XT5 is located at the Arctic Region Supercomputing Center (ARSC) (http://www.arsc.edu/resources/pingo)
  - 432 - Cray XT5 compute nodes with
    - 32 GB of shared memory per node (4 GB per core)
    - 2 - quad-core 2.3 GHz AMD Opteron processors per node.
    - 1 - Seastar2+ Interconnect Module per node.
  - Cray Seastar2+ Interconnect between all compute and login nodes

Cray XT5: CrayPat Performance Analysis

- module load xt-craypat
- Compilation:
  - ftn –fastsse –tp barcelona–64 –r8 –mp=nonuma,[trace ]
- Instrument:
  - pat_build –w –T TraceOmp, –g mpi,omp bt.exe bt.exe.pat
- Execution:
  - (export PAT_RT_HWPC {0,1,2,...})
  - export OMP_NUM_THREADS 4
  - aprun –n NPROCS –S 1 –d 4 ./bt.exe.pat
- Generate report:
  - pat_report –O load_balance,thread_times,program_time,mpi_callers –O profile_pe.th $1

Cray XT5: BT-MZ 32x4 Function Profile

- Results reported for Class D on 256-2048 cores
- SP-MZ pure MPI scales up to 1024 cores
- SP-MZ MPI/OpenMP scales up to 2048 cores
- SP-MZ MPI/OpenMP outperforms pure MPI for 1024 cores
- BT-MZ MPI does not scale
- BT-MZ MPI/OpenMP scales to 2048 cores, outperforms pure MPI

Cray XT5: NPB-MZ Class D Scalability

- Cray XT5: BT-MZ 32x4 Function Profile

- Expected: Load imbalance for pure MPI
- Unexpected!
- Desired: #MPI processes limited

The graphs show the performance scalability of Cray XT5 for NPB-MZ Class D and BT-MZ 32x4 Function Profile, indicating expected and unexpected behavior with respect to load balance and MPI process counts.
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
  - mpixlf_r –O4 –qarch=pwr6 –qtune=pwr6 –qsmp=omp

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

Flag was essential to achieve full compiler optimization in presence of OMP directives.

Conventional Multi-Threading

- 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

NPB-MZ Class D on IBM Power 6:
Exploiting SMT for 2048 Core Results

Doubling the number of threads through hyperthreading (SMT):

```
#!/bin/csh
PBS -l select=32:ncpus=64:mpiprocs=0:ompthreads=NT
```

- Threads alternate
  - Nothing shared

Charles Grassl, IBM
Simultaneous Multi-Threading

- Simultaneous execution
  - Shared registers
  - Shared functional units

Charles Grassl, IBM

Performance Analysis on IBM Power 6

- Compilation:
  - mpirf -O4 -garch=pwr6 -qtune=pwr6 -qomp=omp -pg

- Execution:
  - export OMP_NUM_THREADS 4
  - pree launch SPBS_O_WORKDIR./sp.C.16x4.exe
  - Generates a file gmount(MPI_RANK.out for each MPI Process

- Generate report:
  - gprof sp.C.16x4.exe gmon*

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>total</th>
<th>time</th>
<th>seconds</th>
<th>seconds</th>
<th>calls</th>
<th>ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.7</td>
<td>117.94</td>
<td>117.94</td>
<td>205245</td>
<td>0.57</td>
<td>0.57</td>
<td>.810kx_solve.BLQ8L1 [2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.7</td>
<td>221.14</td>
<td>103.20</td>
<td>205064</td>
<td>0.50</td>
<td>0.50</td>
<td>.815kx_solve.BLQ8L1 [3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.1</td>
<td>307.14</td>
<td>86.00</td>
<td>205200</td>
<td>0.42</td>
<td>0.42</td>
<td>.812kx_solve.OLQ1 [4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>350.83</td>
<td>43.69</td>
<td>205300</td>
<td>0.21</td>
<td>0.21</td>
<td>.8@compute_xha0180L180L86 [5]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Conclusions:

- BT-MZ:
  - Inherent workload imbalance on MPI level
  - #nprocs = #zones 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

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

- Introduction / Motivation
- Programming models on clusters of SMP nodes
- Case Studies / pure MPI vs hybrid MPI+OpenMP
- Practical “How-To” on hybrid programming
  - Georg Hager, Regionales Rechenzentrum Erlangen (RRZE)
  - Mismatch Problems
  - Application categories that can benefit from hybrid parallelization
  - Thread-safety quality of MPI libraries
  - Tools for debugging and profiling MPI+OpenMP
  - 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
  - Intra-node MPI/OpenMP anisotropy
    - MPI communication characteristics
    - OpenMP loop startup overhead
  - Thread/process binding

How to compile, link and run

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

Some examples for compilation and execution (1)

- NEC SX9
  - NEC SX9 compiler
  - mpif90 -C hopt –P openmp ...
  - $ftrace for profiling info
  - Execution:
    $ export OMP_NUM_THREADS=<num_threads>
    $ MPREFIX="OMP_NUM_THREADS"
    $ mpirun -np <# MPI procs per node> -nn np <# of nodes> a.out

- Standard Intel Xeon cluster (e.g. @HLRS):
  - Intel Compiler
  - mpif90 –openmp ...
  - Execution (handling of OMP_NUM_THREADS, see next slide):
    $ mpirun_ssh –np <num MPI procs> –hostfile machines a.out
Handling of OMP_NUM_THREADS

- **without any support by mpirun:**
  - E.g. with mpich-1
  - Problem: mpirun has no features to export environment variables to the via ssh automatically started MPI processes
  - Solution: Set
    
    ```
    export OMP_NUM_THREADS=<# threads per MPI process>
    ```
    
    in ~/.bashrc (if a bash is used as login shell)
  - If you want to set OMP_NUM_THREADS individually when starting the MPI processes:
    - Add `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 (2)

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

Some examples for compilation and execution (3)

- **Sun Constellation Cluster:**
  - `mpif90 -fastsse -tp barcelona-64 -mp ...`
  - SGE Batch System
  - `setenv OMP_NUM_THREADS`
  - `ibrun numaclt.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
  - Downsides: 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
  ```

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 Nehalem)
  - Multi-core, multi-socket
    - Shared vs. separate caches
    - Multi-chip vs. single-chip
    - Separate/shared buses

Issues for running code efficiently on “non-isotropic” nodes

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

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

A short introduction to ccNUMA

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

• CPU
  – 64 kB L1 per core
  – 1 MB L2 per core
  – No shared caches
  – On-chip memory controller (MI)
  – 10.6 GB/s local memory bandwidth
• HyperTransport 1000 network
  – 4 GB/s per link per direction
• 3 distance categories for core-to-memory connections:
  – same LD
  – 1 hop
  – 2 hops

• Q1: What are the real penalties for non-local accesses?
• Q2: What is the impact of contention?

Effect of non-local access on HP DL585 G5:
Serial vector triad \( A(:) = B(:) + C(:) \times D(:) \)

Contenion vs. parallel access on HP DL585 G5:
OpenMP vector triad \( A(:) = B(:) + C(:) \times D(:) \)

ccNUMA Memory Locality Problems

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

- How can we make sure that memory ends up where it is close to the CPU that uses it?
  - See the following slides
- How can we make sure that it stays that way throughout program execution?
  - See end of section

Solving Memory Locality Problems: First Touch

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

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

- It is sufficient to touch a single item to map the entire page

ccNUMA problems beyond first touch

- OS uses part of main memory for disk buffer (FS) cache
  - If FS cache fills part of memory, apps will probably allocate from foreign domains
  - non-local access!
  - Locality problem even on hybrid and pure MPI with "asymmetric" file I/O, i.e. if not all MPI processes perform I/O
- Remedies
  - Drop FS cache pages after user job has run (admin’s job)
    - Only prevents cross-job buffer cache "heritage"
  - "Sweeper" code (run by user)
  - Flush buffer cache after I/O if necessary ("sync" is not sufficient!)

ccNUMA problems beyond first touch

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

- Code (to be run on 2 processors):
  
  ```
  wc = MPI_WTIME()
  do i=1,NREPEAT
    if(rank.eq.0) then
      MPI_SEND(buffer,N,MPI_BYTE,1,0,MPI_COMM_WORLD,ierr)
      MPI_RECV(buffer,N,MPI_BYTE,1,0,MPI_COMM_WORLD, &
                  status,ierr)
    else
      MPI_RECV(…)
      MPI_SEND(…)
    endif
  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)

<table>
<thead>
<tr>
<th>Latency [μs]</th>
<th>IB internode 1S</th>
<th>IB internode 2S</th>
<th>IB internode 3S</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>3,5</td>
<td>0</td>
<td>0.55</td>
<td>0.31</td>
</tr>
</tbody>
</table>

Affinity matters!

IMB Ping-Pong: Bandwidth Characteristics
Intra-node vs. Inter-node on Woodcrest DDR-IB cluster (Intel MPI 3.1)

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

Affinity matters!

OpenMP Overhead

- As with intra-node MPI, OpenMP loop start overhead varies with the mutual position of threads in a team.
- Possible variations
  - Intra-socket vs. inter-socket
  - Different overhead for “parallel for” vs. plain “for”
  - If one multi-threaded MPI process spans multiple sockets,
    - … are neighboring threads on neighboring cores?
    - … or are threads distributed “round-robin” across cores?
- Test benchmark: Vector triad
  ```
  #pragma omp parallel
  for(int j=0; j < NITER; j++){
    #pragma omp (parallel) for
    for(i=0; i < N; ++i)
      a[i]=b[i]+c[i]*d[i];
    if(OBSCURE)
      dummy(a,b,c,d);
  }
  ```

Look at performance for small array sizes!
OpenMP Overhead

Nomenclature:
- 1S/2S: 1-/2-socket
- RR: round-robin
- SS: socket-socket
- inner parallel on inner loop

OMP overhead can be comparable to MPI latency!

Affinity matters!

Thread synchronization overhead

Barrier overhead in CPU cycles: pthreads vs. OpenMP vs. spin loop

<table>
<thead>
<tr>
<th></th>
<th>2 Threads</th>
<th>4 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q9550 (shared L2)</td>
<td>i7 920 (shared L3)</td>
<td></td>
</tr>
<tr>
<td>pthreads_barrier_wait</td>
<td>23739</td>
<td>42533</td>
</tr>
<tr>
<td>omp barrier (icc 11.0)</td>
<td>399</td>
<td>977</td>
</tr>
<tr>
<td>Spin loop</td>
<td>231</td>
<td>1106</td>
</tr>
</tbody>
</table>

Spin loop does fine for shared cache sync

gcc obviously uses a pthreads barrier for the OpenMP barrier:

<table>
<thead>
<tr>
<th></th>
<th>2 Threads</th>
<th>4 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q9550 (shared L2)</td>
<td>i7 920 (shared L3)</td>
<td></td>
</tr>
<tr>
<td>gcc 4.3.3</td>
<td>22603</td>
<td>64143</td>
</tr>
<tr>
<td>icc 11.0</td>
<td>399</td>
<td>977</td>
</tr>
</tbody>
</table>

Correct pinning of threads:
- Manual pinning in source code (see below) or
- likwid-pin: http://code.google.com/p/likwid/

Thread synchronization overhead

Barrier overhead: Topology influence

<table>
<thead>
<tr>
<th></th>
<th>Xeon E5420 2 Threads</th>
<th>Nehalem 2 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>shared L2</td>
<td>same socket</td>
</tr>
<tr>
<td>pthreads_barrier_wait</td>
<td>5863</td>
<td>27032</td>
</tr>
<tr>
<td>omp barrier (icc 11.0)</td>
<td>576</td>
<td>760</td>
</tr>
<tr>
<td>Spin loop</td>
<td>259</td>
<td>485</td>
</tr>
<tr>
<td></td>
<td>Shared SMT threads</td>
<td>shared L3</td>
</tr>
<tr>
<td>pthreads_barrier_wait</td>
<td>23352</td>
<td>4796</td>
</tr>
<tr>
<td>omp barrier (icc 11.0)</td>
<td>2761</td>
<td>479</td>
</tr>
<tr>
<td>Spin loop</td>
<td>17388</td>
<td>267</td>
</tr>
</tbody>
</table>

SMT can be a big performance problem for synchronizing threads
- Well known for a long time...
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 (`OMP_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
  - Widely usable example: Using PLPA under Linux!

Explicit Process/Thread Binding With PLPA on Linux:

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

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 listopo from the hwloc package)
  - likwid-pin: Pin threaded application without touching code
  - likwid-perfCt: Measure performance counters (similar to SGI’s perfex or liotp tools)
  - likwid-features: View and enable/disable hardware prefetchers (Core2 only at the moment)
  - likwid-bench: Low-level benchmark construction tool

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
  - AMD K10 (Quadcore and Hexacore)
  - AMD K8

Output of likwid-topology

CPU name: Intel Core i7 processor
CPU clock: 2668.83826 Hz
******************************************************************************
Hardware Thread Topology
******************************************************************************
Sockets: 2
Cores per socket: 4
Threads per core: 2

-------------------------------------------------------------
HWThread Thread Core Socket
0 0 0 0
1 1 0 0
2 0 1 0
3 1 1 0
4 0 2 0
5 1 2 0
6 0 3 0
7 1 3 0
8 0 0 1
9 1 0 1
10 0 1 1
11 1 1 1
12 0 2 1
13 1 2 1
14 0 3 1
15 1 3 1

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 pthreads, 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 -t intel -c 0,2,4-6 ./myApp parameters`
- `env OMP_NUM_THREADS=6 likwid-pin -c 0:0-2@1:0-2 ./myApp`
- `env OMP_NUM_THREADS=2 mpirun -npernode 2 \
  likwid-pin -s 0x3 -c 0,1 ./myApp parameters`

Example: STREAM benchmark on 12-core Intel Westmere:
Anarchy vs. thread pinning

Topology ("mapping") choices with MPI+OpenMP:
More examples using Intel MPI+compiler & home-grown mpirun

- One MPI process per node
  - `env OMP_NUM_THREADS=8 mpirun -pernode \
    likwid-pin -t intel -c 0-7 ./a.out`
- One MPI process per socket
  - `env OMP_NUM_THREADS=4 mpirun -pernode 2 \
    -pin "0,1,2,3_4,5,6,7" ./a.out`
- OpenMP threads pinned "round robin" across cores in node
  - `env OMP_NUM_THREADS=4 mpirun -pernode 2 \
    -pin "0,1,5,2,3,6,7" \
    likwid-pin -t intel -c 0,2,1,3 ./a.out`
- Two MPI processes per socket
  - `env OMP_NUM_THREADS=2 mpirun -pernode 4 \
    -pin "0,1,3,2_4,5,6,7" \
    likwid-pin -t intel -c 0,1 ./a.out`

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

Mismatches 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 x 16 = 80 sub-domains
- On system with 10 x dual socket x quad-core

Does it matter?

17 x inter-node connections per node
1 x inter-socket connection per node
Sequential ranking of MPI_COMM_WORLD

Never trust the default!!!

The Topology Problem with pure MPI

Application example on 80 cores:

- Cartesian application with 5 x 16 = 80 sub-domains
- On system with 10 x dual socket x quad-core

+ 32 x inter-node connections per node
+ 0 x inter-socket connection per node
Round robin ranking of MPI_COMM_WORLD
The Topology Problem with pure MPI

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
</tr>
<tr>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
</tr>
</tbody>
</table>

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

Two levels of domain decomposition

Bad affinity of cores to thread ranks

The Topology Problem with hybrid MPI + OpenMP

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
</tr>
<tr>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
</tr>
</tbody>
</table>

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

Two levels of domain decomposition

Good affinity of cores to thread ranks

The Topology Problem with

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

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

Successful examples:
- Multi-Zone NAS Parallel Benchmarks (MZ-NPB)

Optimal ?

MPI process 0

Optimal ?

MPI process 1

Loop worksharing on 8 threads

Minimizing ccNUMA data traffic through domain decomposition inside of each MPI process

The Topology Problem with

Application example:
- Same Cartesian application aspect ratio: $5 \times 16$
- On system with 10 x dual socket x quad-core

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
</tr>
<tr>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
</tr>
</tbody>
</table>

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

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 located?
- Which cores share caches?

Case study on Sun Constellation Cluster Ranger with BT-MZ and SP-MZ

Sleeping threads and network saturation with Masteronly

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

Unnecessary intra-node communication

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

Solution:
- Only one MPI process per SMP node

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

Quality aspects of the MPI library
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 master-only 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?

Overlapping Communication and Computation

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

Three problems:
- the application problem:
  - one must separate application into:
    - code that can run before the halo data is received
    - code that needs halo data
  - very hard to do !!!
- the thread-rank problem:
  - comm. / comp. via thread-rank
  - cannot use work-sharing directives
    → loss of major OpenMP support (see next slide)
- the load balancing problem

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

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

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

**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 experiment
- 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.

---

**Overlapping can be achieved with OpenMP tasks (1st part)**

- 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 (2nd 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/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

OpenMP tasking version outperforms original shifter, especially in larger poloidal domains

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.

Intel® Compilers with Cluster OpenMP – Consistency Protocol
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

**Heat example: Cluster OpenMP Efficiency**

- Cluster OpenMP on a Dual-Xeon cluster
  - Efficiency only with small communication footprint
  - Up to 3 CPUs with 3000x3000
  - No speedup with 1000x1000
  - Second CPU only usable in small cases

Comparing results with heat example

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

Back to the mixed model – an Example

- Topology-problem solved: Only horizontal inter-node communication
  - 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
  - Terrible with non-default schedule
  - With Overlapping Comm.& Comp.:
    - Master thread should be reserved for communication only partially – 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 → limited parallelism
    - zones with different workload → speedup < 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
    medium quality cheap implementation
Memory consumption

- Shared nothing
  - Heroic theory
  - In practice: Some data is duplicated
- MPI & OpenMP
  With n threads per MPI process:
  - Duplicated data may be reduced by factor n

Case study: MPI+OpenMP memory usage of NPB

Using more OpenMP threads could reduce the memory usage substantially, up to five times on Hopper Cray XT5 (eight-core nodes).


Always same number of cores

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 m sockets and n cores/socket
- How many threads (i.e., cores) per MPI process?
  - Too many threads per MPI process → overlapping of MPI and computation may be necessary, → some NICs unused?
  - Too few threads → too much memory consumption (see previous slides)
- Optimum
  - somewhere between 1 and m x n threads per MPI process,
  - Typically:
    - Optimum \( \leq n \), i.e., 1 MPI process per socket
    - Sometimes \( \leq n^2 \), i.e., 2 MPI processes per socket
    - Seldom \( \leq 2n \), 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: MPI processes may be multi-threaded, but only master thread will make MPI-calls (virtual value, 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_SERIALIZE: 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!

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

- But this implies that all other threads are sleeping!
- The additional barrier implies also the necessary cache flush!

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>MPICH-2.1.0</td>
<td>ch3:sock supports MPI_THREAD_MULTIPLE</td>
</tr>
<tr>
<td></td>
<td>ch:nemesis has “Initial Thread-support”</td>
</tr>
<tr>
<td>MPICH-2.1.0.8</td>
<td>ch3:nemesis (default) has MPI_THREAD_MULTIPLE</td>
</tr>
<tr>
<td></td>
<td>Intel MPI 3.1</td>
</tr>
<tr>
<td></td>
<td>Full MPI_THREAD_MULTIPLE</td>
</tr>
<tr>
<td>SCI-CORE MPI</td>
<td>SCI-CORE MPI</td>
</tr>
<tr>
<td>HP MPI-2.2.7</td>
<td>Full MPI_THREAD_MULTIPLE (with libmmpi)</td>
</tr>
<tr>
<td>SGI MPT-1.14</td>
<td>Not thread-safe?</td>
</tr>
<tr>
<td>IBM MPI</td>
<td>Full MPI_THREAD_MULTIPLE</td>
</tr>
<tr>
<td>NEC MPI/SX</td>
<td>MPI_THREAD_SERIALIZED</td>
</tr>
</tbody>
</table>

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

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

- In order to enable thread support in Open MPI, configure with:
  ```
  configure --enable-mpi-threads
  ```
- This turns on:
  - Support for full `MPI_THREAD_MULTIPLE`
  - Internal checks when run with threads (`--enable-debug`)
- 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`:
  ```
  Application finished
  ```
<table>
<thead>
<tr>
<th>ID</th>
<th>Short</th>
<th>Severity</th>
<th>Context</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Write -&gt;</td>
<td>Error</td>
<td>pthread_race.c:31 conflicts with pthread_race.c:31</td>
<td>Memory write of global_variable at pthread_race.c:31 conflicts with pthread_race.c:31</td>
</tr>
<tr>
<td></td>
<td>Write da</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Caution: Intel Inspector XE 2011 is a GUI based tool → not suitable for hybrid code execution (?)
Thread Correctness – Intel ThreadChecker 3/3

- If one wants to compile with threaded Open MPI (option for IB):
  ```
  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'
  PFLAGS='-debug all -tcheck' LDFLAGS='tcheck'
  ```

- Then run with:
  ```
  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, delays=2, stall=2'
  ```

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!)
  ```

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

- Summary

Pure MPI – multi-core aware

- Hierarchical domain decomposition
  (or distribution of Cartesian arrays)

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 ➔ not recommended
    - Bottom-up: Low level DD + higher level recombination
  - 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

Example on 10 nodes, each with 4 sockets, each with 6 cores.
Top-down – several levels of (Par)Metis
(not recommended)

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. Socket-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
- In this example the load-balancer must combine always
  • 6 cores, and
  • 4 sockets
- Advantage: Communication is balanced!

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 ...

Weak 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 to 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!

The vendors will (or must) deliver scalable MPI libraries for their largest systems!
Remarks on Cache Optimization

- After all parallelization domain decompositions (DD, up to 3 levels) are done:
  - 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 over these blocks
  - Inner loops inside of a 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 x 5,000 € = 15,000 € (full costs)
- Benefit
  - from reduced CPU utilization
    - e.g., Example 1:
      ```
      100,000 € hardware costs of the cluster
      x 20% used by this application over whole lifetime of the cluster
      x 7% performance win through the optimization
      = 1,400 € \rightarrow \text{total loss} = 13,600 €
      ```
    - e.g., Example 2:
      ```
      10 Mio € system x 5% used x 8% performance win
      = 40,000 € \rightarrow \text{total win} = 25,000 €
      ```

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
  \rightarrow 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?

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
- Summary

Acknowledgements

- We want to thank
  - Co-authors who could not be here for presenting their slides:
    - Gabriele Jost
    - Georg Hager
  - Other contributors:
    - Gerhard Wellein, RRZE
    - Alice Koniges, NERSC, LBNL
    - Rainer Keller, HLRS and ORNL
    - Jim Cownie, Intel
    - KOJAK project at JSC, Research Center Jülich
    - HPCMO Program and the Engineer Research and Development Center Major Shared Resource Center, Vicksburg, MS (http://www.erdc.hpc.mil/index)
Summary – the good news

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

Summary – the bad news

MPI+OpenMP: There is a huge amount of pitfalls
- Pitfalls of MPI
- Pitfalls of OpenMP
  - On ccNUMA → e.g., first touch
  - Pinning of threads on cores
- Pitfalls through combination of MPI & OpenMP
  - E.g., topology and mapping problems
  - Many mismatch problems
- Tools are available 😊
  - It is not easier than analyzing pure MPI programs 😊
  - Most hybrid programs → Masteronly style
- Overlapping communication and computation with several threads
  - Requires thread-safety quality of MPI library
  - Loss of OpenMP worksharing support → using OpenMP tasks as workaround

Summary – good and bad
- Optimization
  - 1 MPI process mismatch 1 MPI process
  - per core ................................................... per SMP node
  - ^ 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

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

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 loosing 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.

Georg Hager

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. A full list of publications, talks, and other HPC-related stuff he is interested in can be found in his blog: http://blogs.fau.de/hager.

Gabriele Jost

Gabriele Jost obtained her doctorate in Applied Mathematics from the University of Göttingen, Germany. For more than a decade she worked for various vendors (Supernum 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 remains in area of performance analysis and evaluation of programming paradigms for high performance computing and that she really liked California. She is now a Research Scientist at the Texas Advanced Computing Center (TACC), working remotely from Monterey, CA on all sorts of exciting projects related to large scale parallel processing for scientific computing.

Book (with direct relation to the content of this tutorial)

References (with direct relation to the content of this tutorial)

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

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

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

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

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

References

- Rolf Rabenseifner,
  Hybrid Parallel Programming on HPC Platforms.

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

- Rolf Rabenseifner and Gerhard Wellein,
  Comparison of Parallel Programming Models on Clusters of SMP Nodes.

- Rolf Rabenseifner and Gerhard Wellein,
  Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures.

References

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

- 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


Further references

- Sergio Bruguglio, 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 DAPSY Conference, Johannes Kepler University Linz, Austria September 29-October 02, 2002
- Nikolaos Drosinos and Nectarios Koziris, Advanced Hybrid MPI/OpenMP Parallelization Paradigms for Nested Loop Algorithms onto Clusters of SMPs, 10th European PVM/MPI Users’ Group Conference (EuroPVM/MPI’03), Venice, Italy, 29 Sep - 2 Oct, 2003

Further references


Further references

Further references


Further references

Hybrid Parallel Programming

Hybrid programming exercises

Jacobi iteration

The Jacobi iteration is a way of solving the Poisson equation $\nabla^2 \phi$ by iteratively updating the value of a 2D array $\phi$ as

$$V_{new}(i,j) = \frac{V(i-1,j) + V(i+1,j) + V(i,j-1) + V(i,j+1) - 4\phi_{i,j}}{4}$$

until convergence has been reached (i.e., $V_{new}$ and $V_{old}$ are sufficiently close to each other).

Parallel Jacobi solver

The update of each element requires only information from the nearest neighbors, therefore if the whole domain can be decomposed to parallel tasks (in either one or two dimensions). Only the boundaries need to be communicated, and one MPI task needs to communicate only with two 1D decomposition or four 2D decomposition other tasks. Therefore the Jacobi solver is decomposed in row-wise (C) or column-wise blocks (Fortran) with the index limits (which depend on the number of MPI tasks and the size of the domain) computed in the procedure “decomp.” Note that two iterations are performed in one cycle of the update loop (to enable reuse of the arrays and more convenient checking for the convergence).

Jacobi solver hybridized

See the Jacobi solver parallelized with MPI based on 2D decomposition of the array $\phi$. Starting from that program, implement an MPI+OpenMP hybrid Jacobi solver with three different realizations:

1. Fine-grained version, where the halo exchange is performed outside the parallel region. This basically means just wrapping the update sweep in a parallel construct.
2. Version where the master thread carries out the halo exchange, and all threads are alive throughout the program execution.
3. Version employing multiple thread communication.

Note, that the hybridization here essentially corresponds to a 2D decomposition. Sample answers are provided in files $\phi$acobi-hyb-a...c.f90 or $\phi$acobi-hyb-a...c.c.

Cray specific remark

When using the Cray systems, you may need to adjust the thread safety of the MPI library with the MPICH_MAX_THREAD_SAFETY variable, e.g., setenv MPICH_MAX_THREAD_SAFETY serialized before a job script. The minimum thread safety level is "single" for Exercise 1, "funneled" for Exercise 2 and "multiple" for Exercise 3. In addition, you will need to link with -lmpich_thread for multiple thread safety support.

Hybrid parallel programming

Practicalities

About compilation

Remember to enable OpenMP when compiling your OpenMP and hybrid programs! With PGI by having the –mp flag, with GNU the –fopenmp flag and so on.

Pure OpenMP jobs

You can run OpenMP enabled codes on a multicore (Linux) laptop simply by setting the environment variable OMP_NUM_THREADS equal to the number of threads you wish to execute, e.g., "export OMP_NUM_THREADS=8", and running the program as usual.

Louhi is not meant as a production platform for flat-OpenMP codes, but for training and testing purposes you can utilize Louhi as well. There an additional -d option to the aprun launcher must be set and the number of MPI processes must be set to one.

Hybrid MPI+OpenMP codes

Hybrid programs are executed in the systems combining the execution of a MPI job and an OpenMP one. On the Cray systems, the execution command

aprun -n 4 -d 8 ./my_hyb.exe

Would launch an interactive job with 4x8=32 cores (4 MPI tasks each having 8 threads).

When executing through the batch job system, additional lines (as compared to a flat MPI program) to the batch script are needed; cf. this sample jobscript

#!/bin/bash
PBS -l walltime=00:15:00
PBS -l mppwidth=16
PBS -l mppdepth=8
PBS -l mppnppn=1
cd $PBS_O_WORKDIR
export OMP_NUM_THREADS=8
aprun -n 16 -N 1 -d 8 ./my_hyb.exe

This would allocate and execute a 16x8=128 core job.