Hybrid (i.e. MPI+OpenMP) applications (i.e. programming) on modern (i.e. multi-socket multi-NUMA-domain multi-core multi-cache multi-whatever) architectures: Things to consider

Georg Hager  Holger Stengel
Gerhard Wellein  Jan Treibig
Markus Wittmann

Erlangen Regional Computing Center (RRZE)
SIAM PP10, MS45
February 26th, 2010
Common lore:

“An OpenMP+MPI hybrid code is never faster than a pure MPI code on the same hybrid hardware, except for obvious cases”

Our statement:

“You have to compare apples to apples, i.e. the best hybrid code to the best pure MPI code”

Needless to say, both may require significant optimization effort.
Outline

- Hybrid programming benefits and taxonomy
  - Vector mode, task mode
  - Thread-core mapping
- “Best possible” MPI code
  - Rank-subdomain mapping
  - Overlapping computation and communication via non-blocking MPI
  - Overlapping cross-node and intra-node communication
- “Best possible” OpenMP code
  - Cache re-use
  - Synchronization overhead
  - ccNUMA page placement
- “Best possible” MPI+OpenMP hybrid code
  - True comm/calc overlap
  - Load distribution issues
  - ccNUMA and task mode

To see it all, visit one of the hybrid tutorials at SC, ISC...!
Hybrid taxonomy and possible benefits
Taxonomy of hybrid “modes”:
Several OpenMP threads per MPI process

Vector mode: MPI is called only outside OpenMP parallel regions. This is what many people mean when they say “hybrid”
- Similar to what we did on vector-parallel machines

Task mode: One or more threads in the parallel region are dedicated to special tasks, like doing communication in the background
- This is functional parallelism on the thread level

### Possible hybrid benefits

<table>
<thead>
<tr>
<th></th>
<th>Vector mode</th>
<th>Task mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improved/easier load balancing</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Additional levels of parallelism</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Overlapping communication and computation</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Improved rate of convergence</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Re-use of data in shared caches</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Reduced MPI overhead</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
Hybrid mapping choices on current hardware
Topology ("mapping") choices with MPI+OpenMP

- One MPI process per node
- One MPI process per socket
- OpenMP threads pinned "round robin" across cores in node
- Two MPI processes per node

02/26/2010 Hybrid Parallel Programming
How do we figure out the topology?

- ... and how do we enforce the mapping?
- Compilers and MPI libs may 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:
  - easy to install
  - works with standard linux 2.6 kernel
  - simple and clear to use
  - supports Intel and AMD CPUs

- Current tools:
  - likwid-topology: Print thread and cache topology
  - likwid-pin: Pin threaded application without touching code
  - likwid-perfCtr: Measure performance counters
  - likwid-features: View and enable/disable hardware prefetchers
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
CPU name:       Intel Core i7 processor
CPU clock:      2666683826 Hz

Hardware Thread Topology

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

02/26/2010 Hybrid Parallel Programming
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 (hybrid)
- Based on combination of wrapper tool together with overloaded pthread library
- Can also be used as replacement for taskset
- Configurable colored output

Usage:
- `likwid-pin -c 0,2,4-6 ./myApp parameters`
- `mpirun likwid-pin -s 0x3 -c 0,3,5,6 ./myApp parameters`
MPI:
Common problems (beyond the usual…)

Rank-subdomain mapping

Overlapping computation with communication
“Best possible” MPI:
Minimizing cross-node communication

Example: Stencil solver with halo exchange

Goal: Reduce cross-node (CN) halo traffic
Subdomains exchange halo with neighbors
- Populate a node's ranks with “maximum neighboring” subdomains
- This minimizes a node's CN communication surface

Shouldn’t MPI_CART_CREATE (w/ reorder) take care of this for me?
MPI rank-subdomain mapping:
3D stencil solver – theory

- "Common" MPI library behavior
- Linear SD distribution
- Optimal SD distribution

Graph showing node communication surface vs. number of cores per node for different platforms:
- Woodcrest 2-socket
- Nehalem EP 2-socket
- Istanbul 2-socket
- Shanghai 4-socket
- Magny Cours 2-socket
- Magny Cours 4-socket
- Nehalem EX 4-socket
- Sun Niagara 2

02/26/2010 Hybrid Parallel Programming
MPI rank-subdomain mapping:
3D stencil solver – measurements for 8ppn and 4ppn GBE vs. IB

32 MPI processes

8 ppn QDR-IB

4 ppn SDR-IB

~ 1.5x
Overlap of computation and non-blocking MPI:
A simple test

- CN communication buffer buf: 80 MB
- do_work() does intra-register work for some amount of time

```c
MPI_Barrier(MPI_COMM_WORLD);
if(rank==0) {
    stime = MPI_Wtime();
    MPI_Irecv/Isend(buf,bufsize,MPI_DOUBLE,1,0,MPI_COMM_WORLD,request);
    delayTime = do_work(Length);
    MPI_Wait(request,status);
    etime = MPI_Wtime();
    cout << delayTime << " " << etime-stime << endl;
} else {
    MPI_Send(buf,bufsize,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
}
MPI_Barrier(MPI_COMM_WORLD);
```
Overlap of computation and non-blocking MPI:
Results for different MPI versions

- Cray XT4 MPT 1.3
- TinyBlue IntelMPI 3.2.2
- TB OpenMPI 1.4 Isend
- TB OpenMPI 1.4 Irecv
- TB mvapich2 1.4rc2

TinyBlue = IBM iDataPlex w/QDR IB

20/02/2010
Hybrid Parallel Programming
MPI take-home messages

- **MPI may not do the best it could when mapping your ranks to your subdomains**
  - Even if all it would take is to know how many processes run on a node

- **MPI may not provide truly asynchronous communication with non-blocking point-to-point calls**
  - Very common misconception
  - Check your system using low-level benchmarks
  - Task mode hybrid can save you 😊
A word about barrier overhead in general...


http://arxiv.org/abs/0910.4865
Thread synchronization overhead

**CUDA vs. OpenMP vs. Spin loop**

<table>
<thead>
<tr>
<th></th>
<th>2 Threads</th>
<th>4 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Q9550 (shared L2)</td>
<td>I7 920 (shared L3)</td>
</tr>
<tr>
<td>pthreads_barrier_wait</td>
<td><strong>23739</strong></td>
<td>6511</td>
</tr>
<tr>
<td>omp barrier (icc 11.0)</td>
<td>399</td>
<td>469</td>
</tr>
<tr>
<td>Spin loop</td>
<td><strong>231</strong></td>
<td>270</td>
</tr>
<tr>
<td></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>Spin loop</td>
<td>1106</td>
<td>475</td>
</tr>
</tbody>
</table>

**Remarks:**
- *Spin loop* does fine for shared cache sync
- *OpenMP & Intel compiler*
Thread synchronization overhead

*OpenMP: icc vs. gcc*

gcc obviously uses pthreads barrier to for OpenMP barrier.

<table>
<thead>
<tr>
<th>2 Threads</th>
<th>Q9550 (shared L2)</th>
<th>I7 920 (shared L3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc 4.3.3</td>
<td>22603</td>
<td>7333</td>
</tr>
<tr>
<td>icc 11.0</td>
<td>399</td>
<td>469</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4 Threads</th>
<th>Q9550</th>
<th>I7 920 (shared L3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc 4.3.3</td>
<td>64143</td>
<td>10901</td>
</tr>
<tr>
<td>icc 11.0</td>
<td>977</td>
<td>814</td>
</tr>
</tbody>
</table>

Correct pinning of threads:

- Manual pinning in source code or
- Prevent icc compiler from pinning → KMP_AFFINITY=disabled
Thread synchronization 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>
</tbody>
</table>

- Spin waiting loops are not suited for SMT
  - Well known for a long time...
  - Roll-your-own barrier may be better than compiler, but take care
Hybrid task mode in action

... and when it makes sense to consider it at all
- 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, $N^3 = 480^3$

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

Performance model:

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

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

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

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

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

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

Performance estimate (GE) for $n$ nodes:

$$P(n) = \frac{N^3}{\left(\frac{T_{\text{COMP}}}{n} + T_{\text{COMM}}(n)\right)}$$
JDS Sparse MVM:
Performance and scalability on two different platforms

The obvious question...

- How do you distribute loop iterations if one thread of your team is missing?
  - Straightforward answer: Use nested parallelism

```c
#pragma omp parallel num_threads(2)
{
  if(!omp_get_thread_num()) {
    // do comm thread stuff here
  }
  else {
    #pragma omp parallel num_threads(7)
    {
      #pragma omp for
      // do work threads stuff here
    }
  }
}
```
However…

- **Nested parallelism must be supported by the compiler**
  - Probably less of a problem today
- **You don’t know what actually happens when starting a new team**
  - ccNUMA page placement?
  - Thread-core affinity?

- **Alternatives:**
  - Use manual work distribution
    - This is somewhat clumsy, but well “wrappable”
    - More importantly, it is static (no advanced scheduling options)
  - Use OpenMP 3.0 tasking constructs
    - Dynamic scheduling (with all its advantages and drawbacks)
    - Communication thread can participate in worksharing activities after communication is over
Conclusions

- **Hybrid MPI+OpenMP programming**
  - is not for the faint of heart
    - Know your basics about NUMA placement, chip/node topology, thread/core affinity
    - Leverage task mode to *really* overlap communication with computation
  - is sometimes unnecessary
    - If pure MPI scales OK, why bother?
  - may give you substantial performance boost
    - But: Try to figure out whether this is possible at all through profiling/tracing and appropriate performance models
THANK YOU
Re-use of shared cache data and relaxed synchronization


Pipelined temporal blocking

- thread 0 ($t_0 \rightarrow t_1$)
- thread 1 ($t_1 \rightarrow t_2$)
- thread 2 ($t_2 \rightarrow t_3$)
Pipelined temporal blocking

One long pipeline (all cores of a node) advances through the lattice, each update is shifted by (-1,-1,-1)

Advantages
- Freestyle spatial blocking
- No explicit boundary copies
- Multiple updates per core

Drawbacks
- Shift reduces cache reuse
- Huge parameter space
- Boundary tiles
Pipelined temporal blocking

*with compressed Grid*

- thread 0 \((t_0 \rightarrow t_1)\)
- thread 1 \((t_1 \rightarrow t_2)\)

**Diagram:**
- Compressed grid
- System grid
Pipelined temporal blocking

- All threads need to synchronize after finishing T iterations on their current tile
- Synchronization gets more expensive with increasing number of threads
Relaxed Synchronisation

- Every thread $t_i$ only increments its own counter $c_i$
- Thread $t_i$ has a minimal distance $d_i$ to its preceding thread $t_{i-1}$
- Thread $t_i$ has a maximal distance $d_u$ to its following thread $t_{i+1}$
- Two threads have at least $d_i$ and at most $d_u$ tiles between them
Performance with different looseness

- Intel Nehalem 2.66 GHz
- 4 cores/socket
- 2 HW threads/core

Graph showing performance with different looseness:
- 2 sockets, 8 cores, 8 threads
- 1 socket, 4 cores, 4 threads

Diagram of processor architecture:
- Core
- L1D
- L2
- L3
- Memory
- QPI
- MC
Performance Results on TinyBlue

Single Node

System size: 600^3