

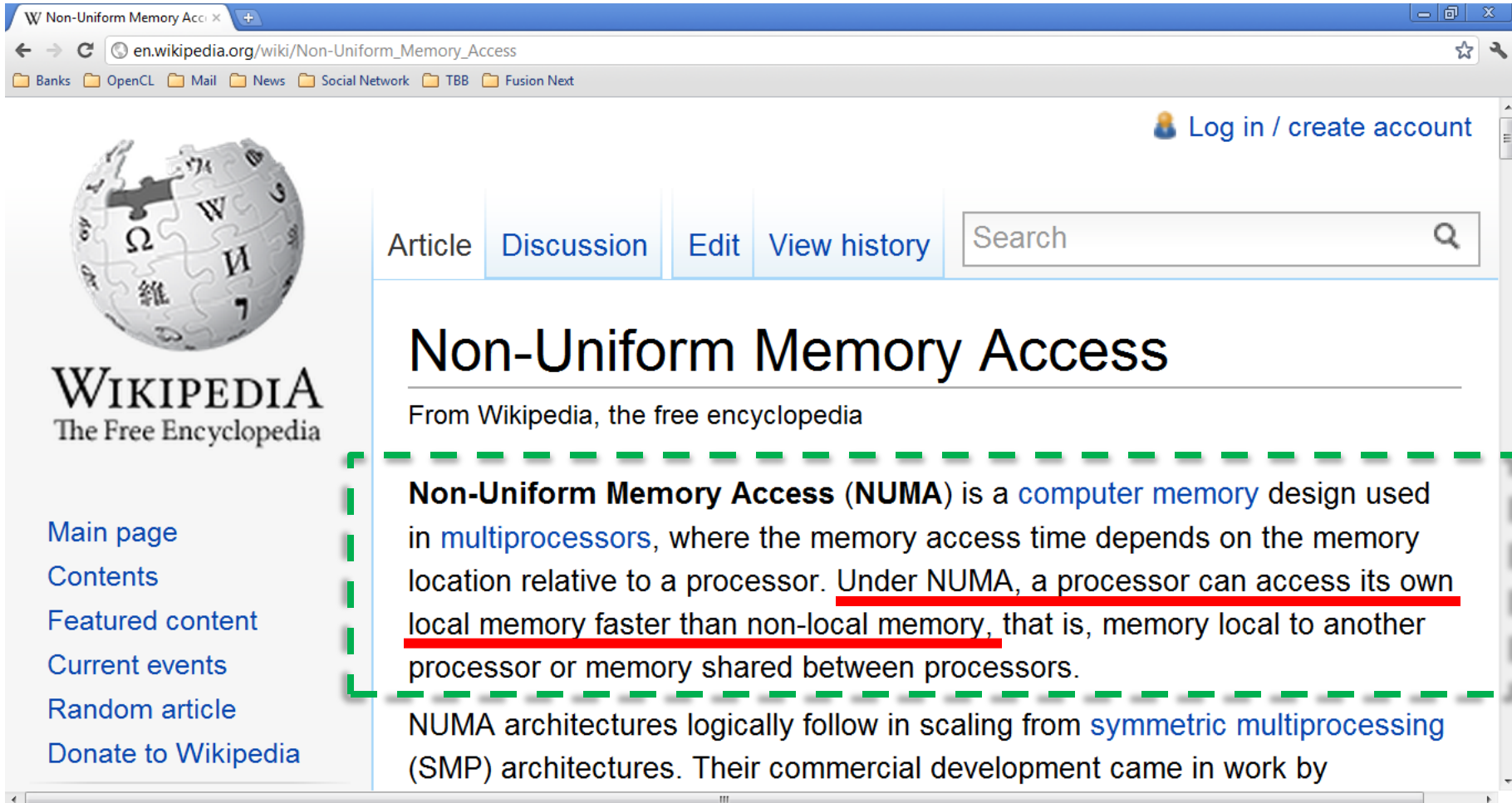
"Best practices for programming with openMP on NUMA systems"

By Joshua.Mora@amd.com

Abstract:

NUMA systems present the challenge to the multithreading software developer on how to allocate data in such a way that memory accesses are maximized with respect to local accesses. Fundamental concepts on the penalties of remote memory accesses versus local memory accesses will be exposed in addition of experimental data collected on AMD based systems. A set of examples will be provided then as a guideline on how to improve the programming of openMP applications with heavy memory access requirements (both memory latency and bandwidth sensitive applications). Additionally, runtime setup is another important component towards the proper exploitation of the NUMA systems with openMP applications. Therefore it deserves to be covered as well within those best practices.

NUMA concepts



W Non-Uniform Memory Acci x

en.wikipedia.org/wiki/Non-Uniform_Memory_Access

Banks OpenCL Mail News Social Network TBB Fusion Next

Log in / create account

Article Discussion Edit View history Search

Non-Uniform Memory Access

From Wikipedia, the free encyclopedia

Non-Uniform Memory Access (NUMA) is a [computer memory](#) design used in [multiprocessors](#), where the memory access time depends on the memory location relative to a processor. Under NUMA, a processor can access its own local memory faster than non-local memory, that is, memory local to another processor or memory shared between processors.

NUMA architectures logically follow in scaling from [symmetric multiprocessing](#) (SMP) architectures. Their commercial development came in work by

Performance metrics

- Summary: It is all about “feeding the beast”
- In order to “crunch” (ie. process) data on the processor you have to feed it with that data. The faster you feed it, the more data it crunches per second.
- FLOP/s is the rate of crunching data on the core, compute unit, processor, node.
- GB/s is the rate of feeding with data (Bytes) that core, compute unit, processor, node.
- FLOP/s and GB/s are related performance metrics

Measuring the feeding rate (GB/s)

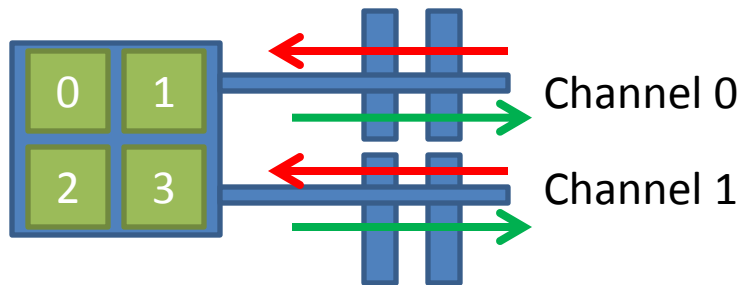
- Single threaded

```
for (i=0;i<1000000;i++) B[i]=A[i] //read A, write B
```

- Multithreaded with openMP

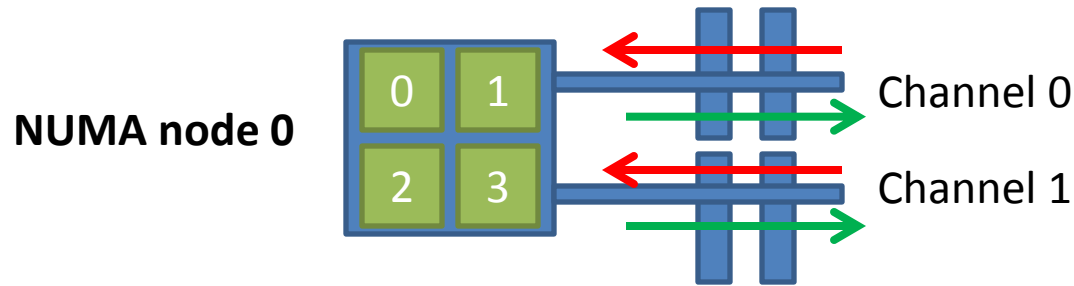
```
#pragma parallel for
```

```
for (i=0;i<1000000;i++) B[i]=A[i] //read A, write B
```

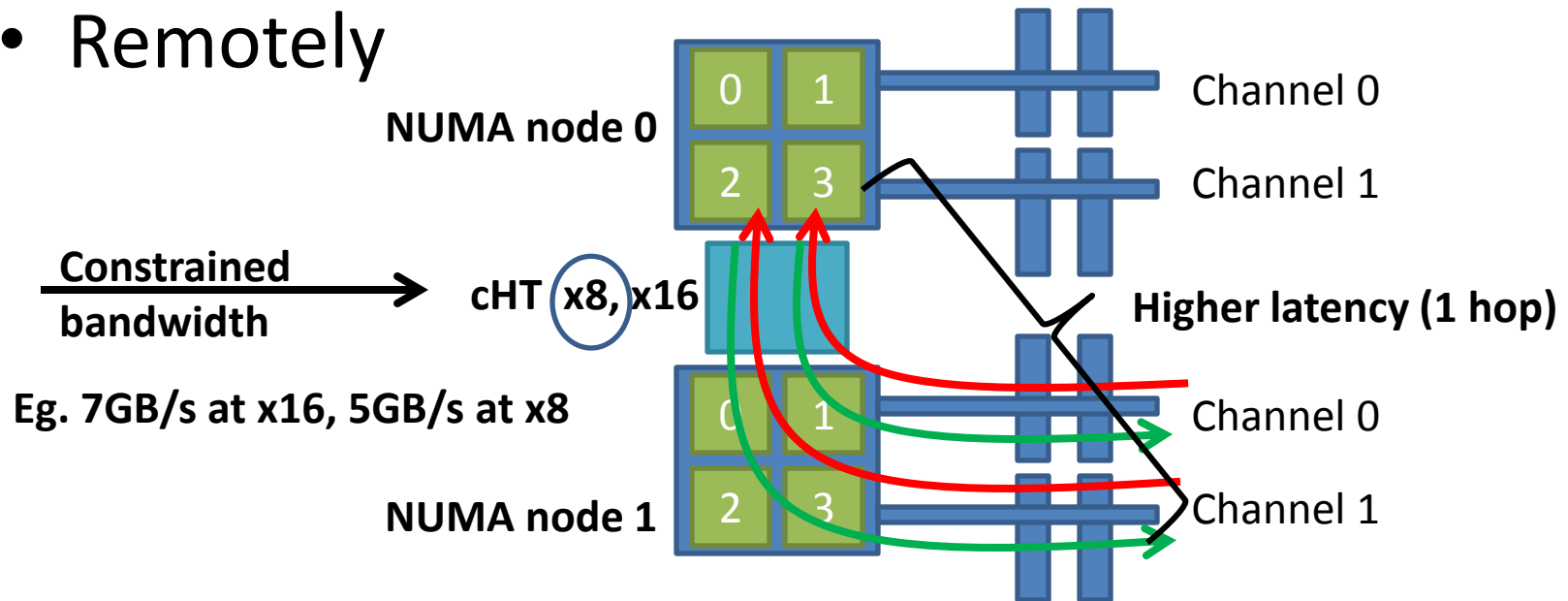


Feeding locally versus remotely

- Locally



- Remotely



Tools to enforce locality

- Fundamental
 - BIOS (Windows, Linux)
- Basic
 - start command, Task Manager (Windows)
 - numactl command (Linux)
 - NUMA APIs (Windows, Linux)
- Advanced
 - HWLOC (Windows, Linux)
 - Likwid (Linux)
 - OpenMP, MPI environment variables (Windows, Linux)

Fundamental, BIOS

- BIOS needs to be set on “Defaults”, ie. F9
- That should set NODE INTERLEAVED DISABLED
- That should set BANK INTERLEAVED ENABLED
- That should set CHANNEL INTERLEAVED ENABLED
- If you “play” with any of those settings, you will **reduce** drastically how you feed the processors.

Question

- How it impacts on performance (feeding the processors) if I set `NODE INTERLEAVED ENABLED` ?

Answer

- Memory pages are “round robin” distributed across all the NUMA nodes. When a core accesses pages for either reads or writes, it incurs into lots of remote memory accesses since they are distributed some locally and mostly remotely.

Example

- Example: A 4 socket G34 with Magnycours, (with 8 NUMA nodes) has 12.5% (1/8) pages accessed at 12GB/s and 87.5% (7/8) pages accessed remotely at 5GB/s through cHT. Aggregated performance with that setting leads to

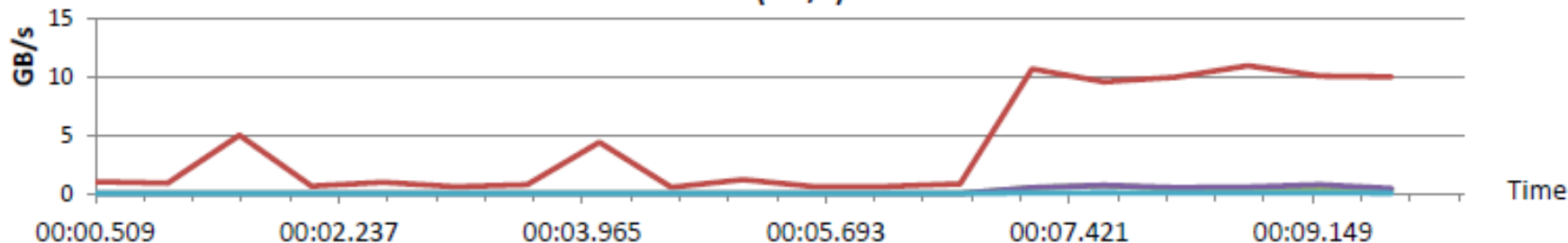
$$\begin{aligned} &8\text{NUMA nodes} * (0.125 * 12\text{GB/s} + 0.875 * 5\text{GB/s}) = \\ &= 8 * 4.375\text{GB/s} = 35\text{GB/s} \end{aligned}$$

But the system is capable of $8 * 12 = 96\text{GB/s}$!!!

Example: OpenMP application on NUMA system

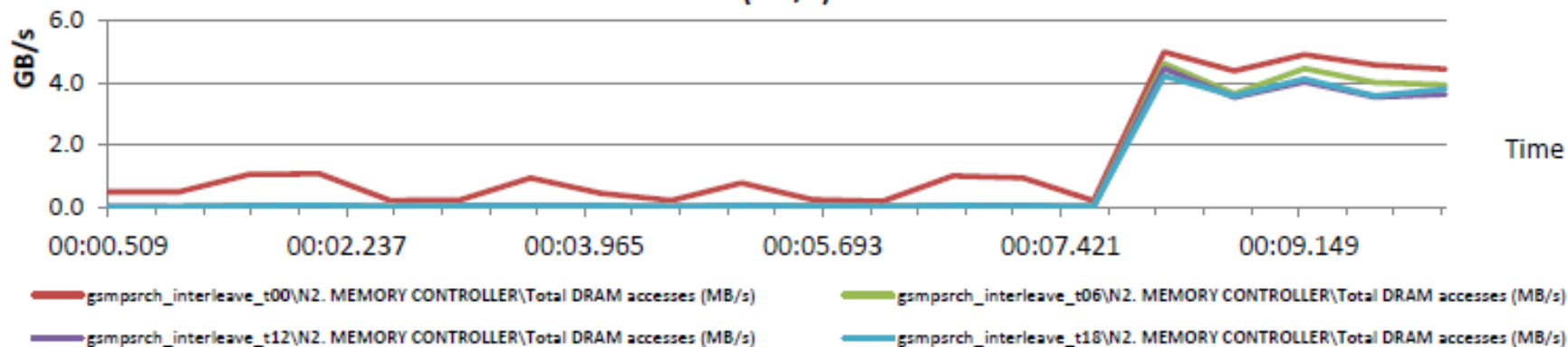
Better to configure system as memory node not interleaved ?

DRAM bw (GB/s) no interleaved



Or memory node interleaved enabled ? Huge cHT traffic, limitation.

DRAM bw (GB/s) interleaved



Answer : modify application to exploit NUMA system by allocating local arrays after threads have started.

Handy tools related with affinity

- “numactl --hardware” to display NUMA nodes, memory available per numa node, cores associated per numa node.
- “numactl [options] ./path/application arguments”
with options: --[physcpu/cpunode]bind=[cores/nodes] ,
--membind=nodes , --interleaved =cores
- “numastat” to display page hits and misses per numa node before and after a run. When a page cannot be allocated on a numa node, it will be allocated on a foreign numa node, therefore incurring into a remote memory access penalty.

NUMA APIs

- Linux

- <http://developer.amd.com/assets/LibNUMA-WP-fv1.pdf>
- http://linux.die.net/man/2/sched_setaffinity

- Windows

MSDN, search for "process and threads functions"

- Process functions, eg. [Get/Set]ProcessAffinityMask
- Thread functions, eg. [Get/Set]ThreadAffinityMask
- NUMA support functions, eg. VirtualAllocExNUMA (we will show how to use this one at the end of the training for a GPU exercise).

HWLOC tool set

- Developed at INRIA

<http://runtime.bordeaux.inria.fr/hwloc/>

- But it can be found under openMPI project

www.open-mpi.org/projects/hwloc

- It is both a set of command line tools (CLI) and an API
 - CLI: lstopo, hwloc-bind, hwloc-ps, ..
 - API: hwloc_[set/get]_[proc]membind_, ..

HWLOC, Istopo

dinar2c01:/opt/hwloc/lib # Istopo

Machine (64GB)

Socket L#0 (32GB)

NUMANode L#0 (P#0 16GB) + L3 L#0 (6144KB)

L2 L#0 (2048KB) + L1 L#0 (16KB) + Core L#0 + PU L#0 (P#0)

L2 L#1 (2048KB) + L1 L#1 (16KB) + Core L#1 + PU L#1 (P#1)

L2 L#2 (2048KB) + L1 L#2 (16KB) + Core L#2 + PU L#2 (P#2)

L2 L#3 (2048KB) + L1 L#3 (16KB) + Core L#3 + PU L#3 (P#3)

L2 L#4 (2048KB) + L1 L#4 (16KB) + Core L#4 + PU L#4 (P#4)

L2 L#5 (2048KB) + L1 L#5 (16KB) + Core L#5 + PU L#5 (P#5)

L2 L#6 (2048KB) + L1 L#6 (16KB) + Core L#6 + PU L#6 (P#6)

L2 L#7 (2048KB) + L1 L#7 (16KB) + Core L#7 + PU L#7 (P#7)

NUMANode L#1 (P#1 16GB) + L3 L#1 (6144KB)

L2 L#8 (2048KB) + L1 L#8 (16KB) + Core L#8 + PU L#8 (P#8)

L2 L#9 (2048KB) + L1 L#9 (16KB) + Core L#9 + PU L#9 (P#9)

L2 L#10 (2048KB) + L1 L#10 (16KB) + Core L#10 + PU L#10 (P#10)

L2 L#11 (2048KB) + L1 L#11 (16KB) + Core L#11 + PU L#11 (P#11)

L2 L#12 (2048KB) + L1 L#12 (16KB) + Core L#12 + PU L#12 (P#12)

L2 L#13 (2048KB) + L1 L#13 (16KB) + Core L#13 + PU L#13 (P#13)

L2 L#14 (2048KB) + L1 L#14 (16KB) + Core L#14 + PU L#14 (P#14)

L2 L#15 (2048KB) + L1 L#15 (16KB) + Core L#15 + PU L#15 (P#15)



HWLOC-ps, where the threads are bound ?

```
bd2:~/streamfinal # export O64_OMP_AFFINITY_MAP=0,2,4,6,8,10,12,14
bd2:~/streamfinal # export OMP_NUM_THREADS=8
bd2:~/streamfinal # ./stream_opencc
```

```
-----
STREAM version $Revision: 5.8 $
-----
This system uses 8 bytes per DOUBLE PRECISION word.
-----
Array size = 50000000, Offset = 0
Total memory required = 1144.4 MB.
Each test is run 50 times, but only
the *best* time for each is used.
-----
```

```
Number of
-----
bd2:/opt/hwloc/bin # hwloc-ps
Printing
Printing 1346 L2Cache:0 L2Cache:2 L2Cache:4 L2Cache:6 L2Cache:8 L2Cache:10 L2Cache:12 L2Cache:14
Printing
Printing ./stream_opencc
Printing
Printing bd2:/opt/hwloc/bin #
Printing
Printing one line per active thread,...
```

```
-----
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 24221 microseconds.
(= 24221 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-----
```

```
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-----
```

Function	Rate (MB/s)	Avg time	Min time	Max time
Copy:	35966.3344	0.0224	0.0222	0.0227
Scale:	35869.8295	0.0226	0.0223	0.0228
Add:	32750.0898	0.0369	0.0366	0.0371
Triad:	32600.7514	0.0371	0.0368	0.0373

```
-----
Solution Validates
-----
```

```
bd2:~/streamfinal # █
```

Process id 1346,
using
0,2,4,6,8,10,12,14
core ids to run stream
benchmark with 8
threads.

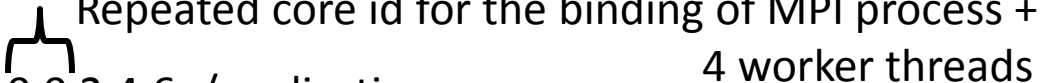
inity"....

LIKWID, for thread binding

- Can be downloaded from <http://code.google.com/p/likwid/>
- It contains the following tools:
- [likwid-topology](#): Show the thread and cache topology
- [likwid-perfctr](#): Measure hardware performance counters on Intel and AMD processors
- [likwid-features](#): Show and Toggle hardware prefetch control bits on Intel Core 2 processors
- [likwid-pin](#): Pin your threaded application without touching your code (supports pthreads, Intel OpenMP and gcc OpenMP)
- [likwid-bench](#): Benchmarking framework allowing rapid prototyping of threaded assembly kernels
- [likwid-mpirun](#): Script enabling simple and flexible pinning of MPI and MPI/threaded hybrid applications
- [likwid-perfscope](#): Frontend for likwid-perfctr timeline mode. Allows live plotting of performance metrics.

EXAMPLE using likwid

Hybrid MPI+OPenMP

- Build application file and launch mpi job with hybrid openMP with 1 thread per compute unit on 2 . Using 4 compute nodes.
 - export OMP_NUM_THREADS=4
 - mpirun -app ./appfile,
 - Where appfile is  Repeated core id for the binding of MPI process + 4 worker threads
- ```
-h node 1 -np 1 likwid-pin -q -c 0,0,2,4,6 ./application
-h node 1 -np 1 likwid-pin -q -c 8,8,10,12,14 ./application
-h node 1 -np 1 likwid-pin -q -c 16,16,18,20,22 ./application
-h node 1 -np 1 likwid-pin -q -c 24,24,26,28,30 ./application
.....
-h node 4 -np 1 likwid-pin -q -c 0,0,2,4,6 ./application
-h node 4 -np 1 likwid-pin -q -c 8,8,10,12,14 ./application
-h node 4 -np 1 likwid-pin -q -c 16,16,18,20,22 ./application
-h node 4 -np 1 likwid-pin -q -c 24,24,26,28,30 ./application
```

# Binding on OpenMP applications

- Imagine we want to spawn 8 threads.  
    `export OMP_NUM_THREADS=8`
- and we want to bind them to core ids 0,2,4,6,8.

For open64 compiler

```
export O64_OMP_AFFINITY_MAP=0,2,4,8
```

For GNU compiler

```
export GOMP_CPU_AFFINITY="0 2 4 8"
```

For PGI compiler

```
export MP_BLIST=0,2,4,8
```

- Then we run the application binary  
    `./application_binary`

CAUTION: do not use `numactl` command on openMP applications. The environment variables listed above assume you are going to do allocation of pages on the numanodes associated with those core ids, therefore maximizing local memory access. `Numactl` would only make sense when you want on purpose to run on interleaved like configurations when there is a lot of remote memory accesses due to lots of shared data among openMP threads.

# Programming with openMP related with local vs remote accesses

## Example 1

- Single thread initializing (writing) data.  
for (i=0;i<1000000;i++) {a[i]=0;b[i]=function();}
- Multiple threads accessing in parallel for reads and writes. Most threads will access data remotely. **Really bad.**

```
#pragma parallel for
```

```
for (i=0;i<1000000;i++) a[i]=b[i];
```

# Programming with openMP related with local vs remote accesses

## Example 1/solution

- Data need to be initialized (written) in parallel as well.

```
#pragma parallel for
```

```
for (i=0;i<1000000;i++) {a[i]=0;b[i]=function();}
```

- threads accessing in parallel for reads and writes. Same threads accessing same data. **Good local access.**

```
#pragma parallel for
```

```
for (i=0;i<1000000;i++) a[i]=b[i];
```

# Programming with openMP related with local vs remote accesses

## Example 2

- Mix of initialized data arrays in serial and parallel. Some arrays in serial (bad), some in parallel(good).

```
for (i=0;i<1000000;i++) a[i]=0;
```

```
#pragma parallel for
```

```
for (i=0;i<1000000;i++) b[i]=function();
```

- All of those data arrays accessed in parallel. **Remote access of serially initialized data slowing down the good parallel and local access .**

```
#pragma parallel for
```

```
for (i=0;i<1000000;i++) a[i]=b[i];
```

- Fix: change those serial initializations to parallel.

# Programming with openMP related with local vs remote accesses

## Example 3

- Mix of initialized/accessed data arrays in parallel with different scheduling schemes.

```
#pragma parallel for ← this default schedule is by blocking
```

```
for (i=0;i<1000000;i++) a[i]=0;
```

```
#pragma parallel for schedule(dynamic)
```

```
for (i=0;i<1000000;i++) b[i]=function();
```

```
#pragma parallel for schedule(dynamic)
```

```
for (i=0;i<1000000;i++) a[i]=b[i]; BAD ACCESSES (lots of remote)
```

- Fix: change those scheduling schemes to be the same.

# Programming with openMP related with local vs remote accesses

## Example 4

- Parallel access of subsets of arrays.

```
#pragma parallel for
```

```
for (i=0;i<1000000;i++) a[i]=0;
```

```
#pragma parallel for
```

```
for (i=0;i<1000000;i++) b[i]=function();
```

```
int halo=1000;
```

```
#pragma parallel for
```

```
for (i=0+halo;i<1000000-halo;i++) a[i]=b[i];
```

- Does each thread still keeps accessing their own portion locally ? Answer: **NO**

# Programming with openMP related with local vs remote accesses

## Example 4/Fix

You have to define **per thread** `initial_index` and `final_index`. Very much like you do with MPI code. Example with 4 threads, array size of 1,000,000 and halo of 1000

|                    | Thread1            | Thread2            | Thread3            | thread4        |
|--------------------|--------------------|--------------------|--------------------|----------------|
| Initial_index      | 0                  | 250,000            | 500,000            | 750,000        |
| final_index        | 250,000 - 1        | 500,000 - 1        | 750,000 - 1        | 1,000,000 - 1  |
| Initial_index_halo | 0                  | 250,000 - halo     | 500,000 - halo     | 750,000 - halo |
| final_index_halo   | 250,000 + halo - 1 | 500,000 - 1 + halo | 750,000 - 1 + halo | 1,000,000 - 1  |

# Programming with openMP related with local vs remote accesses

- Parallel access of subsets of arrays with proper “partitioning of data” **per thread**.

```
#pragma parallel
```

```
for (i=initial_index; i<final_index; i++) {
```

```
 a[i]=0;
```

```
 b[i]=function();
```

```
}
```

```
#pragma parallel
```

```
for (i=initial_index_halo; i<end_index_halo; i++)
```

```
 a[i]=b[i];
```

# Programming with openMP related with local vs remote accesses

- Extend the partitioning in parallel to multiple dimensions to reduce as well long strides on arrays.

```
#pragma parallel
for (k=initial_index_halo_z; k<end_index_halo_z; k++)
 for (j=initial_index_halo_y; j<end_index_halo_y; j++)
 for (i=initial_index_halo_x; i<end_index_halo_x; i++) {
 a[i,j,k]=b[i,j,k]; // here it goes your FD RTM kernel
 }
```

Now, you can explicitly unroll too, to provide clues to the compiler for vectorization, since you know your loop boundaries.

# Programming with openMP related with local vs remote accesses

- Mix the openMP and MPI, to have as many threads as cores in the numanode. Note, AMD processors (Magny-cours and Interlagos) have 2 numanodes per processor. So you don't want to do 12 or 16 threads per process but 6 or 8 threads respectively and 2 MPI processes per processor.
- Build an index for MPI and an index for openMP, paying attention to local accesses. When a halo has to be exchanged, that is done per MPI processes within a numanode.
- By reducing/grouping MPI processes to use threads, it also reduces the MPI message rate, makes it less sensitive to latency, ie. reduces the pressure on the NIC. You'll get better scaling.