Intel® Xeon 5500 Platforms, Integrated Memory Controllers and NUMA

David Levinthal, Julia Fedorova, Dmitry Ryabtsev
SSG/DPD/PAT

* Intel, the Intel logo, Intel Core and Core Inside are trademarks of Intel Corporation in the U.S. and other countries.

Agenda

NUMA and Enabling: Overview
Topology Overview
BIOS Options
OS dependent NUMA concerns
Identifying memory locality (and lack thereof) on Intel® Xeon 5500 processors

Summary
DP Platform dominant validation vehicle

Intel® Xeon™ 5500 Platforms

NUMA, Quickpath and Intel® Xeon™ 5500 Platforms

Quickpath Interfaces greatly increase memory bandwidth of our platforms

- Integrated memory controllers on each socket access dimms
  - Quickpath interconnections provide cache coherency
  - Bandwidth improves by ~4X

Bandwidth improvement comes at a price

- Non uniform memory access
- Latency to dimms on remote sockets is ~2X larger

Peeling away the Bandwidth layer reveals the NUMA Latency layer
NUMA Modes on DP Systems Controlled in BIOS

Non Numa
- Even/Odd lines assigned to sockets 0/1
  - Line interleaving

NUMA mode
- First Half of memory space on socket 0
- Second half on socket 1
- Default on Intel® Xeon™ 5500 Processors

NON-NUMA/NUMA Timings for Specomp* and NAS* Parallel Benchmarks

* Other names and brands may be claimed as the property of others.
Non Uniform Memory Access and Parallel Execution

Process parallel is intrinsically NUMA friendly
- Affinity pinning maximizes local memory access
- MPI
- Parallel submission to batch queues
- Standard for HPC

Shared memory threading is more problematic
- Explicit threading, TBB, openMP*
- NUMA friendly data decomposition (page based) has not been required
- OS scheduled thread migration can aggravate situation

HPC Applications will see Large Performance Gains due to Bandwidth Improvements

A remaining performance bottleneck may be due to non uniform memory access latency

Intel® PTU data access profiling feature was designed to address NUMA
- Intel® Xeon™ 5500 processors events were designed to provide the required data
Data Access Events on Intel® Xeon™ 5500 processors Reveal NUMA Access Pattern

“miss” events are inclusive
- Sum over all data sources and their individual latencies

Intel® Xeon™ 5500 processor Precise events are exclusive

Per data source

Data Access Events Reveal NUMA Access Pattern
Controlling NUMA Data Locality on Linux* and Windows*

Linux* assigns physical pages on “first touch”
- ie buffer initialization not malloc
- If each thread initializes its data, things are good
- Can also use numactl or numalib

Windows assigns physical pages with “allocation”
- VirtualAlloc works like malloc on Linux*
  - Physical pages assigned at first use
- malloc & VirtualAllocExNuma allocation must be parallelized
  - Buffers are no longer contiguous linear address ranges
  - Much MUCH harder

* Other names and brands may be claimed as the property of others.

Data Locality, Threaded Applications and Bandwidth
Consider a threaded triad

```c
int triad(int len, double *a, double *b, double *c, double *x);

int i, bytes = 24;
#pragma omp parallel
{
  #pragma omp for private (i)
  #pragma vector nontemporal
  for(i=0;i<len;i++)a[i]=b[i]+x*c[i];
}
return bytes
```

Parallelizes the work
- function called 1000 times, len=8192000
- ~ 1B cachelines written NT, 2B read
Data Locality, Threaded Applications and Bandwidth

Run an OpenMP* triad under my usual mini_app driver, the resulting BW is only

$\sim 5$bytes/cycle for 8 threads

Running in Non Numa Mode results in

$\sim 8.5$ Bytes/cycle

Why?

**Default Version Allocates Buffers on Thread 0 Using only one Memory Controller**

* Other names and brands may be claimed as the property of others.

Performance Events and NUMA Sources

- **Offcore_Response_0**
  
  8 flavors of Request Type $X$ 8 flavors of $\textit{Sline Source}$
  
  – + all combinations..
  
  ($\sim 65K$ possible programmings)

- One “gotcha”...
  
  NT stores to local Dram appear to go to another core’s cache
  
  (data source = 2 instead of 0x40)
PTU Display Shows Local and Remote Access for OpenMP Triad

Need to Distribute “Allocation” “Allocate” on First Touch

Original allocation

```c
buf1 = (char *) malloc(DIM*(sizeof (double))+1024);
buf2 = (char *) malloc(DIM*(sizeof (double))+1024);
buf3 = (char *) malloc(DIM*(sizeof (double))+1024);
a = (double *) buf1;
b = (double *) buf2;
c = (double *) buf3;
for(num=0;num<len;num++)
{
    a[num]=10.;
b[num]=10.;
c[num]=10.;
}
```

Initialization must also be done in Parallel

* Other names and brands may be claimed as the property of others.
Parallel “Allocation” for Linux*
Requires Parallel Initialization

Parallel allocation

\[
\begin{align*}
buf1 &= (\text{char} *) \text{malloc} (\text{DIM} \times (\text{sizeof (double)}) + 1024); \\
buf2 &= (\text{char} *) \text{malloc} (\text{DIM} \times (\text{sizeof (double)}) + 1024); \\
buf3 &= (\text{char} *) \text{malloc} (\text{DIM} \times (\text{sizeof (double)}) + 1024); \\
a &= (\text{double} *) buf1; \\
b &= (\text{double} *) buf2; \\
c &= (\text{double} *) buf3; \\
\#pragma omp parallel \\
\{ \\
\#pragma omp for private(num) \\
\text{for}(\text{num} = 0; \text{num} < \text{len}; \text{num}++) \\
\{ \\
\quad a[\text{num}] = 10.; \\
\quad b[\text{num}] = 10.; \\
\quad c[\text{num}] = 10.; \\
\} \\
\}
\]=

* Other names and brands may be claimed as the property of others.

**Event Triad:**

<table>
<thead>
<tr>
<th>Event</th>
<th>Triad_omp</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_CLK_UNHALTED.THREAD</td>
<td>2.23E+11</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED.THREAD;Socket 0</td>
<td>7.51E+10</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED.THREAD;Socket 1</td>
<td>1.48E+11</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.ANY_LOCATION</td>
<td>3.13E+09</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.ANY_LOCATION;Socket 0</td>
<td>1.56E+09</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.ANY_LOCATION;Socket 1</td>
<td>1.56E+09</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.LOCAL_CACHE_DRAM</td>
<td>1.56E+09</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.LOCAL_CACHE_DRAM;Socket 0</td>
<td>1.55E+09</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.LOCAL_CACHE_DRAM;Socket 1</td>
<td>80000000</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.REMOTE_DRAM</td>
<td>1.55E+09</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.REMOTE_DRAM;Socket 0</td>
<td>1.55E+09</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.REMOTE_DRAM;Socket 1</td>
<td>1000000</td>
</tr>
</tbody>
</table>

Note socket 0/1 switch between PTU runs
## Event Triad

<table>
<thead>
<tr>
<th>Event</th>
<th>Triad_omp</th>
<th>Triad_NUMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU_CLK_UNHALTED.THREAD</td>
<td>2.23E+11</td>
<td>1.17E+11</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED.THREAD;Socket 0</td>
<td>7.51E+10</td>
<td>5.84E+10</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED.THREAD;Socket 1</td>
<td>1.48E+11</td>
<td>5.83E+10</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.ANY_LOCATION</td>
<td>3.13E+09</td>
<td>3.11E+09</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.ANY_LOCATION;Socket 0</td>
<td>1.56E+09</td>
<td>1.56E+09</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.ANY_LOCATION;Socket 1</td>
<td>1.56E+09</td>
<td>1.55E+09</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.LOCAL_CACHE_DRAM;Socket 0</td>
<td>1.56E+09</td>
<td>3.11E+09</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.LOCAL_CACHE_DRAM;Socket 1</td>
<td>1.55E+09</td>
<td>1.55E+09</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.REMOTE_DRAM;Socket 0</td>
<td>1.55E+09</td>
<td>400000</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.REMOTE_DRAM;Socket 1</td>
<td>1.56E+09</td>
<td>300000</td>
</tr>
<tr>
<td>OFFCORE_RESPONSE_0.ANY_REQUEST.REMOTE_DRAM;Socket 1</td>
<td>100000</td>
<td>100000</td>
</tr>
</tbody>
</table>

---

**5.1 B/cyc vs 8.5 B/cyc vs 12.5 B/cyc on a poorly tuned machine**

---

### OpenMP and Core Affinity Pinning

Export KMP_AFFINITY=compact,0,verbose will pin affinity of threads

Just not reproducibly (per socket) on Red Hat 5.1 from run to run

Causing problems in multi run PTU collections

Problem is that an app does not use OMP runtime libs to pin affinity until there is a #pragma parallel {}

You must add this around first instruction to pin affinity of Main thread
Multi-thread Scaling and NUMA

When measuring scaling between 4 and 8 threads (assuming no SMT) the affinity of the 4 threads matters

4 threads all on one socket has the same LLC cache size/core as 8 threads

BUT

2 threads/socket has closer to the same memory BW as the 8 thread run

Thus 4->8 scaling will always have a non scaling contribution due to one of these 2 effects

Per Socket Display + Data Source events
Show NUMA /Cross Socket Traffic
Indirect Addressing, Locality and Latency (Diff Eq on Non Uniform Grid, Oil Res)

Multi-dimensional array access can cause large address gaps in data decomposition.

This can make mapping NUMA home node->pages->data decomposition ranges challenging.

Ex: color = decomposition = thread

64.5K Structures

Default Initialization Breaks Array into 8 Contiguous Pieces ➞ 50% Non Local Access
Address Histogram for all Dram Accesses

Filtering to a Single Thread Displays the Data Decomposition
A Different Thread

Using Only Precise Remote Dram Event
Only Half the entries shown
Gaps due to lack of events are suppressed
Using Only Precise Remote Dram Event
Only Half the entries shown
Gaps due to lack of events are suppressed

Change Initialization to Follow Work Access Pattern
Thread initialization with same access sequence as work
Expect ~33% improvement
  - 1/2 of accesses get lower latency by 2
Simple OMP ran in 14.3 cycles/cell
NUMA initialized version ran in 11.2 cycles/cell
Every access has serious DTLB issues, which don’t change with the improved NUMA layout
Sampling View for Correctly Initialized Array has no Remote Access

Page Allocation Order Matters

Serially initialized/allocated
Accessed with complex pattern
avg Lat =230

Initialized/Allocated
and Accessed with complex pattern
avg Lat = 209
Conclusions

NUMA will add complexity to software performance analysis and optimization

We have the infrastructure to manage this

Backup