Thread and Memory Placement on NUMA Systems: Asymmetry Matters

Baptiste Lepers, Alexandra Fedorova (Simon Fraser University), Vivien Quéma (Grenoble INP)

ATC 2015
Introduction

Current threads and memory placement: minimizing hop-count (e.g. in Linux).

Contributions:

- Connections are asymmetric, bandwidth is more important than hops.
- AsymSched algorithm that dynamically places threads and memory.
Inter-node bandwidths for 4 AMD Opteron 6272 processors

Figure 1: Modern NUMA systems, with eight nodes. The width of links varies, some paths are unidirectional (e.g., between 7 and 3) and links may be shared by multiple nodes. Machine A has 64 cores (8 cores per node - not represented in the picture) and machine B has 48 cores (6 cores per node). Not shown in the picture: the links between nodes 4 and 1 and between nodes 2 and 7 are bidirectional on machine B. This changes the routing of requests from node 7 to 2 and node 1 to 4.

Figure 2: Performance difference between the best, and worst thread placement with respect to the average thread placement on Machine A. Applications run with 24 threads on three nodes. Graph500, specjbb, streamcluster, pca and facerec are highly affected by the choice of nodes and are shown separately with a different y-axis range.

Figure 3: Difference in latency of memory accesses between the best, and worst thread placement with respect to the average node configuration on Machine A. Positive numbers mean that memory accesses are faster than the average.
Measurements

Applications running on 3 nodes, with different node placements.
streamcluster running on 2 nodes, with different node placements.

<table>
<thead>
<tr>
<th>Master thread node</th>
<th>Execution Time (s)</th>
<th>Diff with 0-1 (%)</th>
<th>Latency of memory accesses (cycles) (compared to 0-1(%)</th>
<th>% accesses via 2-hop links</th>
<th>Bandwidth to the &quot;master&quot; node (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>148</td>
<td>0%</td>
<td>750</td>
<td>0</td>
<td>5598</td>
</tr>
<tr>
<td>-</td>
<td>228</td>
<td>56%</td>
<td>1169 (56%)</td>
<td>0</td>
<td>2999</td>
</tr>
<tr>
<td>0</td>
<td>228</td>
<td>56%</td>
<td>1179 (57%)</td>
<td>0</td>
<td>2973</td>
</tr>
<tr>
<td>2</td>
<td>168</td>
<td>15%</td>
<td>855 (14%)</td>
<td>0</td>
<td>4329</td>
</tr>
<tr>
<td>0</td>
<td>340</td>
<td>133%</td>
<td>1527 (104%)</td>
<td>98</td>
<td>1915</td>
</tr>
<tr>
<td>3</td>
<td>185</td>
<td>27%</td>
<td>1040 (39%)</td>
<td>98</td>
<td>3741</td>
</tr>
<tr>
<td>0</td>
<td>340</td>
<td>133%</td>
<td>1601 (113%)</td>
<td>98</td>
<td>1903</td>
</tr>
<tr>
<td>5</td>
<td>228</td>
<td>56%</td>
<td>1206 (61%)</td>
<td>98</td>
<td>2884</td>
</tr>
<tr>
<td>3</td>
<td>185</td>
<td>27%</td>
<td>1020 (36%)</td>
<td>0</td>
<td>3748</td>
</tr>
<tr>
<td>7</td>
<td>338</td>
<td>132%</td>
<td>1614 (115%)</td>
<td>98</td>
<td>1928</td>
</tr>
<tr>
<td>1</td>
<td>338</td>
<td>132%</td>
<td>1612 (115%)</td>
<td>98</td>
<td>1891</td>
</tr>
<tr>
<td>5</td>
<td>230</td>
<td>58%</td>
<td>1200 (60%)</td>
<td>0</td>
<td>2880</td>
</tr>
<tr>
<td>2</td>
<td>167</td>
<td>15%</td>
<td>867 (16%)</td>
<td>98</td>
<td>3748</td>
</tr>
<tr>
<td>7</td>
<td>225</td>
<td>54%</td>
<td>1220 (63%)</td>
<td>0</td>
<td>3014</td>
</tr>
<tr>
<td>4</td>
<td>230</td>
<td>58%</td>
<td>1205 (60%)</td>
<td>0</td>
<td>2959</td>
</tr>
<tr>
<td>1</td>
<td>226</td>
<td>55%</td>
<td>1203 (60%)</td>
<td>98</td>
<td>2880</td>
</tr>
</tbody>
</table>
AsymSched

- User-level thread+memory placement manager
- Continuously measures communication
- Decides every second whether threads/memory should be migrated
AsymSched – Measurement

- Reads some hardware counter (data accesses from CPU to node)
- No counter for CPU to CPU available
- Assumes for decision making:
  - Threads on same node share data
  - Between nodes with 'high' communication threads of same application share data.
AsymSched – Decision

- Puts threads of same application that share data into clusters.
- Each cluster gets weight
  \( C_w = \log(\#\text{remote memory accesses}) \).
- For each placement (mapping of clusters to nodes), compute
  \( P_w = \sum_{C \in \text{Clusters}} C_w \cdot (\text{max bandwidth for } C) \).
- Select placements whose \( P_w \geq 90\% \) of maximal \( P_w \). Of those choose that with least page migrations.
- If cost for memory migration (assuming 0.3s per GB) is too high, do not apply placement.
- Because of symmetry, not all placements need to be tested. Also “obviously bad” placement are ignored.
AsymSched – Migration

- Uses *dynamic* (lazy) *migration*.
- If after 2 seconds > 90% of accesses go to old node, do full migration.
- Full migration uses special system call, that is faster than *migrate_pages*, because it stops the application and needs less locks.

<table>
<thead>
<tr>
<th>Migrated memory (GB)</th>
<th>cg.B</th>
<th>ft.C</th>
<th>is.D</th>
<th>sp.A</th>
<th>streamcluster</th>
<th>graph500</th>
<th>specJBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average time - Linux syscall (ms)</td>
<td>860</td>
<td>12700</td>
<td>101000</td>
<td>490</td>
<td>750</td>
<td>1500</td>
<td>50500</td>
</tr>
<tr>
<td>Average time - fast migration (ms)</td>
<td>51</td>
<td>380</td>
<td>3050</td>
<td>30</td>
<td>45</td>
<td>90</td>
<td>1500</td>
</tr>
</tbody>
</table>
AsymSched

A few exceptions are is.D, lu.B and kmeans. For is.D, the impact on performance is not visible because of the bursty; placing its threads has a huge impact on the latency closely matches that of the performance.

The applications studied in section 5.2. We chose two different clustering configurations in which the links used by streamcluster are not shared with any other applications. AsymSched

We evaluate several multi-application workloads using the applications that benefit from thread and memory migration alone exhibits high standard deviation and, like single application workloads, is unable to improve performance for Graph500 and SPECjbb. This is because in both cases, the applications benefit (to a small degree), and matrixmultiply (does not benefit). Some of these applications have different properties before emulating a three-tier client/server system.

Figure 6 presents the performance on multi application workloads. We chose two different clustering configurations: (i) Three applications executing on three, three and two nodes, respectively; (ii) Two applications executing on five and three nodes respectively.

Figure 5: Memory latency under the best and worst static thread placement, dynamic memory placement, and the average thread placement on machine A. Applications run with 24 threads on 3 nodes.

AsymSched

Figure 5 shows the latency of memory accesses compared to average placement (%)

Worst Placement Dynamic Memory Placement Only AsymSched

Worst placement Dynamic Memory Placement Only AsymSched

Worst placement Dynamic Memory Placement Only AsymSched

Latency of memory accesses compared to average placement (cycles)

Performance – 1 application on 3 nodes

Evaluation – 1 application on 3 nodes

Worst Placement Dynamic Memory Placement Only AsymSched

Worst placement Dynamic Memory Placement Only AsymSched

Worst placement Dynamic Memory Placement Only AsymSched

Graph500

SPECjbb

pca

facerec

streamcluster

wmem

wr

kmeans

swaptions

us A+x

mg C+x

lu B+x

ls D+x

fs C+x

ff C+x

sg C+x

bg B+x

bu B+x

l

Perf. improvement relative to average placement (%)
Evaluation – 3 applications

AsymSched uses our custom system call (1.5% overhead). To keep the overhead low, we only use the standard Linux system call (50% overhead), but only 3 seconds are needed. For is.D, migration takes 101 seconds using the standard Linux system call (see Section 4.3). Table 5 compares the migration time when running the standard Linux system call and using fast memory migration.

The main overhead of AsymSched is due to memory migration. In the worst case, for is.D, migration takes 101 seconds using the standard Linux system call and using fast memory migration.

The overhead of thread migration is negligible and we did not observe any noticeable effect of thread migrations on cache misses.

The cost of collecting metrics and computing clusters is about 2MB of RAM.

In practice, the maximum migration overhead we observed was 3%. If the predicted overhead is below 5%, in practice, the placement constraints are likely to be possible on future machines with a much larger number of nodes. With very simple heuristics we were able to avoid computing up to 99% of the possible thread placements. Such optimizations will still likely be possible on future NUMA machines.

We believe that the findings and the solution presented in this paper can scale on machines, as machines are usually made of multiple identical cores/sockets (e.g., our 64-core machine has 4 identical sockets). On machines that offer a wider diversity of cores/sockets, as machines are usually made of multiple identical cores/sockets.

First, we believe that the clustering and placement techniques used in AsymSched can scale on machines with very high numbers of nodes. With very simple heuristics we were able to avoid computing up to 99% of the possible thread placements. Such optimizations will still likely be possible on future NUMA machines.
Discussion

- What’s the matter with memory migration?
- How well would this work without the magic constants?
- What if #threads is not a multiple of #cores in NUMA-domain?