NumaMMA
NUMA MeMory Analyzer

François Trahay – Télécom SudParis
Manuel Selva – INRIA
Lionel Morel – CEA
Kevin Marquet – INRIA
NUMA architectures are now common

- Multi-socket systems
- Multicore CPUs
  - AMD Infinity Fabric (Zen CPU family)
  - Intel sub-NUMA clustering (Skylake family)
- Increase the available memory bandwidth
Impact of locality

- **NUMA** = Non Uniform Memory Access
  - Fast access to the local memory
  - Slower access to remote NUMA nodes

→ the locality of memory access impacts the performance
  eg. impact on NPB LU on a 48-core machine: up to 27%
→ need to allocate pages on the right NUMA node
**First-touch policy**

- Default policy on Linux
- Lazy allocation policy
- Allocate a page locally when a thread touches it
  - Assumption: this thread is probably the one that will use the page the most
    - Assumption may be wrong!

```c
double *array = malloc(sizeof(double)*N);
for(int i=0; i<N; i++) {
    array[i] = something(i);
}
#pragma omp parallel for
for(int i=0; i<N; i++) {
    double value = array[i];
    /* ... */
}
```
Memory allocation strategies
Interleaved

**Interleaved policy**
- The pages are allocated on multiple nodes in round-robin
  → Balance the load on multiple NUMA nodes

```c
double *array = numa_alloc_interleaved(sizeof(double)*N);
for(int i=0; i<N; i++) {
    array[i] = something(i);
}
#pragma omp parallel for
for(int i=0; i<N; i++) {
    double value = array[i];
    /* ... */
}
```
Memory allocation strategies
Manual placement of memory pages

- Manual placement of memory pages
  - Move pages to a specific NUMA node with mbind() or move_pages()
    + precise control of page placement
  - manual placement of each page

```c
double *array = malloc(sizeof(double)*N);
for(int i=0; i<N; i++) {
    array[i] = something(i);
}
mbind(&array[0], N/4*sizeof(double), MPOL_BIND, &nodemask, maxnode, MPOL_MF_MOVE);
#pragma omp parallel for
for(int i=0; i<N; i++) {
    double value = array[i];
    /* ... */
}
Choosing the best binding policy

- The threads access pattern may differ from one object to another
  - eg. Matrix multiplication

- The best policy for one object may degrade the performance for another object

- Is it worth finding the best policy for one object?
  - Some objects are rarely accessed
NumaMMA – NUMA MeMory Analyzer

- Analyze the memory access pattern of parallel applications
  - Low overhead collection of memory access

- Report:
  - The most accessed memory objects
  - Which thread access which part of an object
  - The evolution of access patterns over the time

- Freely available as open-source
  - https://github.com/numamma
NumaMMA
Collecting memory access information

**Hardware memory sampling**

- eg. Intel PEBS, AMD IBS
- Every X instructions, the CPU collects a sample
  - Address of the memory load/store
  - Thread/Instruction that issued the memory access
  - Where the data is stored (cache, RAM, remote RAM, …)
  - Cost of the memory access (ie. latency)

- Information stored in a buffer
- Low overhead sampling (eg. < 1-2 %)

```
#tid  timestamp   address      mem_level     latency
0     5835423725112 0x557735cf07c8 L2 Hit     24
0     5835456302591 0x557736353ef8 Local RAM Hit 779
0     5835466068752 0x55773642a0c0 Local RAM Hit 657
0     5835471131886 0x5577362726e8 L2 Hit     23
0     5835566865010 0x557735d1fd28 L3 Hit     52
0     58355567586835 0x557735d04710 L3 Hit     64
0     583554540592 0x55773620900 Local RAM Hit 1585
0     5835605025940 0x557735c39900 Local RAM Hit 265
0     5835618194705 0x557735f0e428 L2 Hit     24
0     5835693753719 0x557735f16a78 L2 Hit     23
0     5835709318658 0x557736260f00 Local RAM Hit 266
```
NumaMMA
Identifying memory objects

- **Static memory object**
  - eg. global variables
  - Search for symbols in the ELF binary

- **Dynamic memory objects**
  - malloc, realloc, calloc, free, …
  - Intercept dynamic allocations with LD_PRELOAD

- **For each object, NumaMMA knows**
  - The allocation/de-allocation timestamp
  - The start/end address
NumaMMA
Matching samples

- For each sample, find the matching memory object
- Once found, update counters
  - Number of read/write accesses
  - Total cost of memory accesses to the object
- For large objects, counters are computed per page
- Generate a summary of the most accessed objects

Summary of the call sites:
----------------------------------------
Sorting call sites
0  fields_ (size=2520000) - 34098 read access (total weight: 362881, avg weight: 10.642296). 66541 wr_access
1  [stack] (size=412316860415) - 47982 read access (total weight: 345827, avg weight: 7.207432). 60001 wr_access
2  constants_ (size=1272) - 589 read access (total weight: 5131, avg weight: 8.711375). 0 wr_access
3  /usr/lib/x86_64-linux-gnu/libgomp.so.1(+0x9b49) [0x7f6b06eb4b49] (size=192) - 96 read access (total weight: 672, avg weight: 7.347222)
NumaMMA
Memory access patterns

- Access pattern to an object
  - X-axis: threads
  - Y-axis: memory pages

- Thread 0 access pattern
  - Pages [0-992] [4267-5176]

- Thread 3 access pattern
  - Pages [993-1969] [5178-6074]

- Due to sampling, some pages are not detected
  - Depends on the sampling frequency
NumaMMA
Evolution of access patterns over the time

- Access pattern to an object over the time
  - X-axis: time
  - Y-axis: memory pages
  - Color: thread id

- Allows to detect the phases of the application

![Graph showing access patterns over time with X-axis as time, Y-axis as memory pages, and different colors for thread ids.](image-url)
Evaluation

- Run multithreaded applications from NAS Parallel Benchmark and Parsec
  - Evaluate the overhead
  - How to use NumaMMA to improve the execution time of applications?

Experiment setup

- Intel32:
  - 2 Intel Xeon E5-2630 v2 CPUs (16 cores/32 threads)
  - 32 GiB RAM (2 NUMA nodes)
  - Linux 4.11, GCC 6.3

- AMD48:
  - 4 AMD Opteron 6174 CPUs (48 cores)
  - 128 GiB RAM (8 NUMA nodes)
  - Linux 4.10, GCC 6.3
### Evaluation

#### Overhead

**Overhead on NAS Parallel Benchmarks**
- Running on Intel32
- OpenMP Implementation
- Threads are bound

**2 settings**
- **NumaMMA 2K**
  - Sampling rate: 2000
  - Samples collected during malloc/free
- **NumaMMA 10K**
  - Sampling rate 10,000
  - Sample collection every 100ms

<table>
<thead>
<tr>
<th>kernel</th>
<th>native time(s)</th>
<th>NumaMMA_2k time(s)</th>
<th>ovhd(%)</th>
<th>nsamples</th>
<th>NumaMMA_10k time(s)</th>
<th>ovhd(%)</th>
<th>nsamples</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT.C</td>
<td>81.3</td>
<td>82.3</td>
<td>1.30</td>
<td>0.7 M</td>
<td>85.7</td>
<td>5.46</td>
<td>172 M</td>
</tr>
<tr>
<td>CG.C</td>
<td>21.6</td>
<td>23.3</td>
<td>8.01</td>
<td>0.7 M</td>
<td>22.2</td>
<td>2.93</td>
<td>41 M</td>
</tr>
<tr>
<td>EP.C</td>
<td>10</td>
<td>10.5</td>
<td>3.81</td>
<td>0.6 M</td>
<td>10.5</td>
<td>4.40</td>
<td>19 M</td>
</tr>
<tr>
<td>FT.C</td>
<td>19.6</td>
<td>20.2</td>
<td>3.14</td>
<td>0.5 M</td>
<td>21.7</td>
<td>10.77</td>
<td>39 M</td>
</tr>
<tr>
<td>IS.C</td>
<td>1.5</td>
<td>1.48</td>
<td>-2.11</td>
<td>0.18 M</td>
<td>1.45</td>
<td>-4.35</td>
<td>3 M</td>
</tr>
<tr>
<td>LU.C</td>
<td>61.8</td>
<td>58</td>
<td>-6.11</td>
<td>0.65 M</td>
<td>61.7</td>
<td>-0.12</td>
<td>110 M</td>
</tr>
<tr>
<td>MG.C</td>
<td>10.4</td>
<td>11.3</td>
<td>9.01</td>
<td>0.8 M</td>
<td>10.9</td>
<td>5.48</td>
<td>22 M</td>
</tr>
<tr>
<td>SP.C</td>
<td>168.6</td>
<td>169.8</td>
<td>0.68</td>
<td>0.5 M</td>
<td>169.6</td>
<td>0.59</td>
<td>334 M</td>
</tr>
<tr>
<td>UA.C</td>
<td>86.6</td>
<td>93.5</td>
<td>8.00</td>
<td>0.6 M</td>
<td>96.5</td>
<td>11.37</td>
<td>171 M</td>
</tr>
</tbody>
</table>

→ **Low overhead**
→ **High resolution of a subset vs. Low resolution of the whole application**
Evaluation
Case study: NPB LU

Application: NPB LU
- LU matrix factorization
- OpenMP Implementation

Most accessed objects:

<table>
<thead>
<tr>
<th>symbol</th>
<th>size</th>
<th>nb read</th>
<th>nb write</th>
<th>total</th>
<th>percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvar</td>
<td>558 MB</td>
<td>112k</td>
<td>118k</td>
<td>230k</td>
<td>35.4</td>
</tr>
<tr>
<td>stack</td>
<td>?</td>
<td>105k</td>
<td>124k</td>
<td>229k</td>
<td>35.2</td>
</tr>
<tr>
<td>cexact</td>
<td>520 B</td>
<td>86k</td>
<td>0</td>
<td>86k</td>
<td>13.2</td>
</tr>
<tr>
<td>cjac</td>
<td>20 MB</td>
<td>39k</td>
<td>54k</td>
<td>94k</td>
<td>14.4</td>
</tr>
<tr>
<td>libgomp.so.1(+0x97e9)</td>
<td>8 KB</td>
<td>4k</td>
<td>97</td>
<td>4k</td>
<td>0.7</td>
</tr>
</tbody>
</table>

Access pattern analysis
- cexact is accessed evenly by the threads → no easy optimization
- cvar and cjac could be optimized
Evaluation
NPB LU: Analyzing the access pattern

- **cvar**
  - 558MiB buffer
  - Accessed by slices of 160MiB
  → block-cyclic distribution

- **cjac**
  - 20MiB buffer
  - Accessed by slices of 5MiB
  → block-cyclic distribution

Access pattern to cvar

Access pattern to cjac
Evaluation
NPB LU: optimizing memory placement

Evaluation on AMD48

Comparing different memory placement

- First-touch: default policy
- Interleaved: interleave the memory pages of cvar and cjac
- Block-naive: use a block distribution for cvar and cjac
- NumaMMA: place cvar and cjac using a block-cyclic distribution

<table>
<thead>
<tr>
<th>policy</th>
<th>execution time(s)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>first-touch</td>
<td>102.53</td>
<td>1</td>
</tr>
<tr>
<td>interleaved</td>
<td>106.86</td>
<td>0.96</td>
</tr>
<tr>
<td>block-naive</td>
<td>109.88</td>
<td>0.93</td>
</tr>
<tr>
<td>NumaMMA</td>
<td>81.05</td>
<td>1.27</td>
</tr>
</tbody>
</table>

→ 27% performance improvement
## Evaluation

### Case study: Streamcluster

#### Application: Parsec Streamcluster

#### Most accessed objects

- **block** – 98 MiB buffer (66% of the samples)
  - Evenly distributed accesses
    → interleave pages

- **points** – 6 MiB buffer (31% of the samples)
  - Each thread accesses a part of the buffer
  → block distribution

#### Evaluation

<table>
<thead>
<tr>
<th>policy</th>
<th>execution time(s)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>first-touch</td>
<td>93.72</td>
<td>1</td>
</tr>
<tr>
<td>interleaved</td>
<td>76.76</td>
<td>1.22</td>
</tr>
<tr>
<td>block-naive</td>
<td>79.75</td>
<td>1.17</td>
</tr>
<tr>
<td>NumaMMA</td>
<td>73.32</td>
<td>1.28</td>
</tr>
</tbody>
</table>

→ 28 % improvement
Conclusion & future work

- Memory placement is important for performance
- **NumaMMA: NUMA MeMory Analyzer**
  - Use hardware sampling to collect memory access samples
  - Report the most accessed memory objects
  - Report the threads access pattern over the time
  - Available as open-source: [https://github.com/numamma/numamma](https://github.com/numamma/numamma)

**Evaluation**
- Low overhead (< 12%)
- Reported information can be used for improving performance by up to 28%

**Future work**
- Port over AMD cpus
- Use a signal to handle overflows
- Automate memory placement