Scientific approaches: NUMA Profilers/analyzing runtime behavior
Non-Uniform Memory Access (NUMA) Seminar

Malte Swart
Hasso-Plattner-Institut

10 December 2014
1 Motivation

2 Improvable memory access patterns

3 Existing Profiles
   - SNPERF
   - NumaTOP
   - vTune
   - MemProf
   - MemAxes

4 Summary

5 References
Motivation

**Problem**

We have our new powerful NUMA system. But our application does not scale as it does on UMA systems. What can we do?
Motivation (2)

What can we do?

Upgrade the kernel: We have already a current kernel – so no automatic improvement by new scheduling techniques . . .

Look at the source code: We did not write the application, so no real change for improvement there

Using performance counter: as we see in the last presentation

Analyze our program with profilers: Let’s do it . . .
## What is achievable?

- We concentrate on remote memory accesses, local caches mostly irrelevant in comparison / not NUMA specific
- Next: identify common problems that we can optimize and need to identify
Remote usage after allocation

**Issue**
Data is created on one NUMA node, but only used on another

**Solutions**
- Create data directly on other node
- Copy data on first access (if copying is amortized)
- Migration thread to node with data

Figure: Remote usage after allocation
Alternate remote accesses to an object

Issue
Data is read by multiple NUMA nodes, but only from one at a time (concurrent but not parallel)

Possible Solutions
- Pin threads to NUMA node with their data
- Migrate threads over time to their data

Figure: Alternate remote accesses to an object
Parallel remote accesses to an object

**Issue**

Parallel access by multiple NUMA nodes

**Solution**

- Duplicate data, if not or rarely changed (more memory needed)
- Move on thread to the other node (might result in load imbalance)

**Figure**: Concurrent remote accesses to an object
Two common types of profiles (example after [1]):

1. instruction/code-orientated profiles (line 4: 100% latency)
2. data-oriented profiles (Array A: line 4: 1% latency; Array B: line 4 - 10% latency, Array C: line 4 - 89% latency)

```c
for (int i=0; i < n; i++) {
    for (int j=0; j < n; j++) {
        for (int k=0; k < n; k++) {
        }
    }
}
```

Traditional profiles concentrate on cache optimization and code hot spots
Existing Profiles (2)

Problem
Information about code or data itself is less useful
UMA profilers not very helpful for NUMA issues

Challenge
Thread on Node x accessed data from Node y
→ Profilers with more information are needed
SNPERF - a ccNUMA Profiling Tool

- One of the earliest NUMA profiler (Developed around 2001)
- Designed for Origin2000 systems
- Basic on simple performance counters to measure memory bandwidth saturation
Motivation

Improvable memory access patterns

Existing Profiles

Summary

References

SNPERF Examples (1)

Sniper Examples (1)

Figure 6: FFT memory utilization profile on four nodes

Points referred to as the complex roots of unity. Both sets are organized as $p \times p$ matrices, and the matrices are partitioned so that every processor is assigned a continuous set of rows that are allocated in its local memory.

Interprocessor communication is limited to the transpose steps.

Figure 6 shows the memory utilization profile on four nodes for the entire duration of an 8-processor FFT run. The first phase of the execution is the initialization of the roots of unity and the generation of random input data. The initialization is performed on the main thread, which runs on processor 0. The memory utilization profiles for each node show that the data set was distributed equally among the four nodes—the plot from start to 3 sec shows first utilization on node 0, then node 1, node 2, and node 3; the utilization has two peaks on each node, the first time when the application initializes the data for the roots of unity matrix and the second time when it generates the random data.

The second phase shows the progress of the FFT algorithm. The three peaks in the memory utilization plot correspond to the matrix transposes. These are the interprocessor communication phases, where every processor transposes a portion of the data matrix. The two valleys in between correspond to the 1-D FFT transformation on each (local) row and the application of the roots of unity. The barriers before the second and third transpose are visible as the sharp drops in memory utilization.

The transpose algorithm used by the SPLASH-2 FFT kernel works in two phases: first, each processor transposes a patch (contiguous submatrix) of size $p \times p$ from every other processor, and then transposes the patch locally. The transpose takes advantage of long cache lines by blocking. The original SPLASH-2 FFT uses staggering to communicate patches between processors: processor $i$ first transposes a patch from processor $i+1$, then from processor $i+2$, etc., to prevent hotspotting. If the processors move in lockstep, no two processors read from the same processor's patch of memory at the same time. We will call this communication pattern the basic stagger. However, there are no barriers inside the SPLASH-2 FFT transpose algorithm. It is entirely possible that one or more processors fall behind the others, because it was preempted by system activity, for example. Since the processors transpose patches in a sequential manner, one delayed processor could cause a domino effect, and further delay other processors that follow it.

To avoid this scenario, a second transpose algorithm uses a binary scrambling function to compute the next processor whose patch is to be transposed; this is the optimized stagger algorithm. Both staggered transposes are contrasted with the naive matrix transpose where each processor first replaces a patch from processor 0, then processor 1, and so on. This is the unoptimized transpose algorithm.

Figures 7–9 show high-resolution memory utilization profiles for unoptimized transpose, basic, and optimized staggering, respectively. All figures show the second transpose step in a 16-processor run for a data set size of 4 million elements; each run assigned two threads to each node, allocating memory on 8 nodes. The memory utilization is shown for even-numbered nodes only.

Not surprisingly, the unoptimized transpose algorithm results in memory hotspots: as the processors transpose patches, they first overrun the memory capacity on node 0, then node 1 and so on. The basic stagger eliminates

Figure: FFT memory utilization profile on four nodes
SNPERF Examples (2)

Figure: Unoptimized FFT matrix transpose without staggering
SNPERF Examples (3)

Figure: FFT matrix transpose with optimized staggering
SNPERF Examples (4)

Figure: FFT matrix transpose with optimized staggering
NumaTOP

Question

Does we have a NUMA problem (high remote memory access)?
And no poorly scaling application

NumaTOP

- Live ranking between different running tasks
- Measures local/remote memory access for different processes / nodes
- Some special view about stats of NUMA nodes or memory ranges
**NumaTOP Demo**

![Example output of NumaTOP](image)

**Figure**: Example output of NumaTOP

**RMA(K)**: remote (non-local NUMA node) memory accesses (in 1000)

**LMA(K)**: local memory accesses (in 1000)

**RMA/LMA**: remote memory percentage - should be low

**CPI**: CPU cycles per instruction
vTune

- Specialized profiler from Intel
- Based on performance counters
- But more traditional profiler (cache misses, % operation stalled)

“If [remote memory] percentage is significant (>20%), consider strategies for improving NUMA access: use a NUMA-aware memory allocator, privatize variables. System tuning: ensure memory is balanced across nodes.” [2]

Problem
Generic advices, no hints what to do exactly with a given problem.
Challange

Combine remote memory access with detailed information about allocation and object properties

→ Generate flow graphs for objects and threads

1. Execute program and dump information about object and thread lifecycle and memory accesses
2. Generate flow graphs (offline - after execution)
MemProf: Object lifecycle tracking

**Own dynamic library**
- Overrides memory management functions (like `malloc`) – stores profile information and calls original library
- Needs to be loaded manual per `LD_PRELOAD` or `dlsym`
MemProf: Thread lifecycle tracking

Own dynamic library

- Overloaded kernel functions `pref_event_task` and `perf_event_comm0`
MemProf: Memory access tracking

- Microarchitecture profiling technique
- "Instruction Based Sampling" by AMD, "Precise Event Based Sampling" is similar technique by Intel (PMU technique)
- processor selects single instructions on a given frequency
- interrupt containing information about instruction used to process the data
- random based approach → variation of results
MemAxes

- Similar profiler like MemProf
- Aggregate profile information from performance monitoring units (PMUs)
- Gathers also information about hardware topology (caches, NUMA nodes . . .)
MemAxes: Semantic Annotations

Semantic Annotations
Developer decides which data structures are interested to profile
Developer can optimal aggregate additional attributes

Listing 1: Profile matrix A

```c
#define N 1024
double A[N][N]; // matrix data object

SMRTree *smrt = new SMRTree();
SMRNode *A_SMR = smrt->addSMR("A", sizeof(double), A, N*N);
```
MemAxes: Semantic Annotations (2)

Listing 2: Aggregate further application specific fields

```c
// smrt is from previous example
smrt->addIntegerAttribute("x_coord",-1);
smrt->addIntegerAttribute("y_coord",-1);

void* mat_attrtribution(SMRNode *smr, struct mem_sample *sample)
{
    // Obtain the index of the address
    int bufferIndex = smr->indexOf(sample->daddr);
    // Calculate the x and y indices (row-major)
    sample->setAttribute("x_coord", bufferIndex % N);
    sample->setAttribute("y_coord", bufferIndex / N);
}
```
Motivation Improvable memory access patterns

Existing Profiles

Summary References

MemAxes: Working Principle

Figure: Basic working principle of MemAxes
MemAxes: Visualization

**Figure**: Visualization principle of memAxes; left: hardware topology in general; right: with cpu latency (color) and usage (line width)
**Motivation**

**Improvatable memory access patterns**

**Existing Profiles**

**Summary**

**References**

---

**MemAxes: Example**

[Diagram of MemAxes example]

**Figure**: Application with unoptimized affinities
MemAxes: Example (2)

Figure: Same application with optimized affinities
Tips for algorithm groups

- Profilers support identifying the cause of intensive remote memory access
- NumaTOP is a good start (easily installable, check whether we have a NUMA problem)
- vTune could give some hints in what direction to look
- MemProf not usable as it currently depends on AMD profiling instructions
- MemAxes is the most powerful tool, but take a little bit more time to use it (smaller code adjustments) and I was unable to find the tracing sources itself
Summary

- Runtime behavior of NUMA applications is difficult to understand / predict
- Concrete tracing information needed to identify issues
- Automatic tools (e.g. kernel scheduling) not always able to produce optimal placing
- Profilers have currently only limited support for identifying NUMA issues
- NUMA profilers are an active research and development field
- Even with this information may it be complicated to optimize your application
- NUMA problems remain performance problems
Images are extracted from the corresponding papers!

  
  MemProf source code. https://github.com/Memprof

References (2)


- NumaTOP v1.0 Documentation ([https://01.org/sites/default/files/documentation/numatop_introduction_0.pdf](https://01.org/sites/default/files/documentation/numatop_introduction_0.pdf))

