A tool for profiling memory accesses locality on NUMA architectures

Letícia S. F. Machado[1], **Hermes Senger**[1], **Claude Tadonki**[2]

[1]Departamento de Computação - Universidade Federal de São Carlos (UFSCar)
São Carlos - SP - Brazil

[2]Centre de Recherche en Informatique (CRI) - Mines ParisTech - PSL France
Fontainebleau, France

suellenletici@gmail.com, senger.hermes@gmail.com

claude.tadonki@minesparis.psl.eu

Abstract. *In this paper, we studied the NUMA architecture and memory access patterns presented by applications executing in this architecture. We developed a tool to parse the source code of a target application and produce traces about which memories of the NUMA node were accessed and in which order. Based on these traces, we expect to get insights into strategies to optimize the performance of the target applications. We will apply this tool to improve the performance of a wave equation simulator which is the kernel of a seismic imaging application for the Oil and Gas industry.*

1. Introduction

This paper investigates high-performance computing (HPC) solutions for efficient geophysical exploration. This technique is considered enabling technology for improving the efficiency of the Oil and Gas industry in the coming years. Seismic imaging techniques are extensively used in geophysical exploration. For example, the full-waveform inversion (FWI) and the reverse time migration (RTM) are essential applications for the identification and placement of hydrocarbon reservoirs and the characterization of the subsurface material like porosity, viscosity, acoustic velocity, localization, dimensions, and others. Furthermore, FWI and RTM workflows are known to be computationally heavy. Even though the processing power of modern processors is increasing, their memory systems are increasingly complex. Mapping stencil codes efficiently on such processors is complicated and is still under intensive study. We focus on the non-uniform memory access (NUMA) model since most large multi-core processors are of NUMA type. To improve performance, we focus on improving memory access, stencil behavior with the NUMA architecture, where it is stored, and how to use this information efficiently. For this purpose, we created a generic tool that works with any source code with similar memory access.

2. NUMA Architectures

Regarding shared memory parallelization, the case of NUMA processors requires special attention because of the particular memory organization and the impact on the overall performance. The NUMA configuration was designed to alleviate the bottleneck scenario where all CPU cores use the same unique bus to access the main shared memory, thereby maintaining a high probability of good scalability over many cores. Unfortunately, good scalability can be obtained only if all memory accesses are local. Indeed, remote accesses are more costly because of the additional mechanism to convey the data and the contention on the QPI links and the concerned NUMA node. Kaestle et al. [Kaestle et al. 2015] propose a library for parallel programs on NUMA machines based on array abstraction and memory allocation routines, which allows automatic tuning of data placement and accesses for better scalability. Lin et al. [Lin et al. 2016] propose efficient stencil computations using many-core NUMA architectures, targeting higher performance and portability. On NUMA architectures, the cores ideally should limit access to local memory positions, i.e., memory positions placed in the same NUMA node.

3. Source code application profiling

To analyze the memory access patterns of our stencil code, we need to instrument the source code of our application to produce the necessary traces. To read the source code and generate other code on top of it in the way we wanted, we used Flex, a lexical analyzer generator. The lexical analyzer recognizes lexical patterns in a text. In this way, when passing source code, Flex analyzes the description created by the user in regular expressions and C code, which are called rules [Brown et al. 1992]. When running the executable, the tool will analyze the entries for occurrences of regular expressions. Whenever it finds one, it executes the corresponding C code [Paxson et al. 2007]. We chose to use the *Flex* tool with *lex* due to the extensive documentation on how to use it and the ease and simplicity of visualizing its functionality. To create the parsing, we also had to understand how the information we wanted would be collected. In this context, `numactl` and `libnuma` are two essential tools for controlling NUMA aspects as we need. Numactl runs processes with a specific NUMA scheduling or memory placement policy, which is valid by all of its children. In addition, it can set persistent policies for shared memory segments or files. Also, the libnuma library is included in the numactl package, offering a simple programming interface to the NUMA policies supported by the kernel.

4. A tool for source code analysis

The type of memory access we are interested in this study follows the format `A[B]`, where `A` is the base address of a vector, and `B` is a number or expression that denotes an offset (or an index). After identifying this type of memory access in the source program, our tool transforms this access into `func(numa_node_of_cpu(sched_getcpu()), A, B)`, where in this function, the NUMA node of CPU and the NUMA node of the memory where `A[B]` is stored are taken, without changing the final result of the source code. Another functionality of our code is the ability to compress all the data and transforms it in a more understandable information. For instance, one possible base code:

```
for(j=0; j< N; j++){
    X[i] = A[j*i]*X[j];
}
```

Working with our parser, focusing on `A[j*i]`, the instrumented code would look as follows:

```
for(j=0; j< N; j++){
```

Offset range	CPU node
13:1714	1

Table 1. CPU node output.

Offset range	MEM node
13:13	0
13:13	1
14:14	0
14:14	1
...:...	...
1714:1714	0
1714:1714	1

Table 2. MEM node output.

```
        X[i] = func(numa_node_of_cpu(sched_getcpu()), A, j*i)*X[j];
    }
```

5. Experimental evaluation

To display our tool, we analyzed the stencil code that simulates the wave equation available at `https://github.com/HPCSys-Lab/wave-equation`, focusing on the locality regarding the memory of the NUMA nodes. First, we illustrate the outputs produced by our tool in the form of several .txt files as shown in Tables 1 thru 4. For this example, we chose to instrument the source code that simulates the wave equation in 3D domain. In these tables, we used small values for illustration. Then, the repeated accesses are summarized. Next, given the data we collected, we show in which NODE was the majority of the MEM node and CPU node accesses, in Table 2 and in Table 1. The CPU is always in NUMA node 1, which implies that, as shown in the Table 2, every time that MEM node is stored in NUMA node 0, that access is being performed less efficiently than those in 1. In addition, it is important to analyze which NUMA node in general was most accessed, disregarding whether it was CPU or MEM. This result is shown in table 3. In this particular situation, given the previous information, ideally there should be no access at all in NUMA node 0, but this is not the case.

Furthermore, since the NUMA node that the CPU is in necessarily needs to access the MEM node to fetch the information it is looking for, we need to analyze the trace and say how many times this search is being done within the same NUMA node or in others. In the example of Table 4, we have only 2 NODE nodes, and the quantity tells how many times NODE 0 accessed NODE 0 and 1 and how many times NODE 1 accessed NODE 0 and 1. So, according to the result in table 4, only 30894 accesses are being done efficiently.

NUMA node	Quantity
0	39106
1	100894

Table 3. Frequency output.

NUMA node accesses	Quantity
$0 \rightarrow 0$	0
$0 \rightarrow 1$	0
$1 \rightarrow 0$	39106
$1 \rightarrow 1$	30894

Table 4. Matrix of access output.

5.1. Discussion on how to use our profile reports

After understanding what each output of the instrumented code produces, the next step is understanding what these results mean for NUMA architectures. For example, in Table 3 we see that NUMA node 0 was accessed as many times as NUMA node 1. In this case, analyzing also the results of the Table 4, we can see that all the accesses we had were of the type NUMA node 0 on NUMA node 1. Considering everything that has already been said about the NUMA architecture in Section 2, it is noticeable that this type of access is inefficient in terms of speed since the access is much faster when it occurs within the same NUMA node.

6. Conclusion

In summary, we proposed a method to help the programmer to produce traces of the memory usage in NUMA devices. To evaluate our method, we implemented a tool that provides useful information, such as how often NUMA nodes are used, a matrix that shows the access from one NUMA node to another, and other features. For this reason, it can be said that we accomplished our initial goal. Our parsing tool is available in `https://github.com/HPCSys-Lab/NUMA-metric`.

As the next steps, we intend to use our method and the implemented tool to propose metrics that could quantify the locality of access of an application executing on NUMA architectures. With this metric, we expect to get insights on how to improve the memory locality of applications.

Acknowledgements

This work results from a Research Internship Abroad at the Mines ParisTech University, France, funded by grant #2022/11070-9, São Paulo Research Foundation (FAPESP). The authors are grateful for the support of FAPESP (grant # 2019/26702-8).

References

Brown, D., Levine, J., and Mason, T. (1992). *Lex & yacc*. " O'Reilly Media, Inc.".

Kaestle, S., Achermann, R., and Roscoe, T. (2015). Shoal: smart allocation and replication of memory for parallel programs. In *USENIX Annual Tech. Conf.*, page 8–10.

Lin, P., Yi, Q., Quinlan, D., Liao, C., and Yan, Y. (2016). Automatically Optimizing Stencil Computations on Many-core NUMA Architectures. In *International Workshop on Languages and Compilers for Parallel Computing*.

Paxson, V., Estes, W., and Millaway, J. (2007). Lexical analysis with flex. *University of California*, page 28.