# A Source-to-Source NUMA Profiling Approach

Leticia Suellen Farias Machado, Claude Tadonki, Hermes Senger

2023 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW) | 979-8-3503-8160-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/SBAC-PADW60351.2023.00018

# A Source-to-source NUMA Profiling Approach

Leticia S.F. Machado
Computer Science Department
UFSCar - São Carlos, SP, Brazil
suellenletici@gmail.com

Claude Tadonki
Centre de Recherche en Informatique (CRI)
Mines ParisTech - PSL, Fontainebleau, France
claude.tadonki@mines-paristech.fr

Hermes Senger
Computer Science Department
UFSCar - São Carlos, SP, Brazil
suellenletici@gmail.com

***Abstract*—The design of HPC processors is driven by the purpose of packaging an increasing number of CPU cores. This trend in the multicore design faces the physical reality of integrating circuits into a single die in addition to the bottleneck of components sharing, thus the advent of *Non-Uniform Memory Access* (NUMA) with its typical packaging. Cutting-edge supercomputers are made up of such (manycore) compute nodes. In any case, the main issue is scalability. With a NUMA configuration, a memory access can be *local* (within the same NUMA node) or *remote* (from a NUMA node to another). The latter is the main concern w.r.t to efficiency because of the associated overhead is much more important. Dealing with this concern explicitly when designing a program is called *NUMA-aware implementation*. With an existing code, the problem can be addressed by starting with an appropriate profiling. This is the focus of the present work, where we suggest a way to instrument the native code in order to get the type (i.e. *local* or *remote*) of each memory access and we provide a tool that supports the profiling process. We then propose a metric that takes these statistics about memory accesses and provides a value indicating the potential associated overhead.**

## I. Introduction

This study was motivated by the investigation of High Performance Computing (HPC) solutions for efficient geophysical exploration. Seismic imaging applications are considered major enabling technologies to improve the efficiency of the Oil and Gas industry in the coming years [1]. 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 also for the characterization of the subsurface material like *porosity, viscosity, acoustic velocity, localization, dimensions*, and others. Stencil patterns can be fine-grained for point-wise computations like in image processing or coarse-grained for macroscopic schemes like what can be found with simulation codes in experimental physics [2]. Furthermore, FWI and RTM workflows are known to be computationally heavy. Typically, the execution of an FWI scenario can take several months on a Petaflop/s cluster, with data collected within the range from 2 to 10 Hz. Moreover, the computational cost of FWI will keep increasing significantly in the coming years because of the availability of better quality data (from new acquisition technologies), the need for higher resolution images (i.e., processing higher frequency data), and the need to shorten the "time to first oil" and improve the industry efficiency. Even though the processing power of modern processors is increasing, their memory systems are increasingly complex. At the level of a compute node, the typical parallelization model is the *shared-memory* one since we are dealing multicore processors, some of them having a NUMA configuration.

Non-Uniform Memory Access (NUMA) configuration stands as the standard approach to build up large multicore processors in a modular way. Modulary here includes the multi-socket characteristic that allows to extend the number of available cores associated to the same processor. Note that the basic idea behind NUMA packaging was to alleviate the pressure on the (unique) shared bus as can be seen with symmetric shared memory multiprocessor (SMP). A NUMA processor is like a shared memory cluster of multicore processors, each of which having its "local" memory while having access to that of the others ("remote" memory). Remote accesses are costly for mainly two reasons: the non-linear path from the core to the remote memory and a potential saturation of the interconnect between the NUMA nodes. The latter is a typical consequence of a NUMA-unaware scheduling, especially when more cores belonging to distinct nodes are involved. This is why we commonly see a clear stagnation of the speedup curve with benchmarks on NUMA processors, thus the need of so-called *NUMA-aware* designs where the aforementioned aspects are taken into account at the level of *memory allocation* or at the level of *tasks management*. An example of NUMA-aware design is provided in the work by Tadonki on *Lattice Quantum ChromoDynamics* (LQCD) [2] with an illustration of its impact on the overall parallel efficiency. Many other examples are available in the litterature [3], [4], [5], [6], [7]. In case NUMA-awareness is to be applied in a given ordinary parallel code, a relevant profiling is clearly needed in order to figure out the memory access pattern according to the NUMA configuration. This is the purpose of our work, where we propose a *source-to-source* approach and its associated implementation as a *profiling tool*.

Our idea is to build the *NUMA memory access pattern* by recording for each access the triplet: (virtual) memory address, NUMA node of its physical location, NUMA node of the thread requester. The code is instrumented through a high-level transformation where each array access is replaced by a function that returns the expected value while provisioning the aforementioned memory access trace. With this trace that is provided at the end of the profiling execution, the programmer can investigate how to reduce the number of *remote accesses*. In this work, we have (i) designed a profiling methodology; (ii) built a tool to perform the corresponding high-level transfor-

979-8-3503-8160-3/23/$31.00 ©2023 IEEE
DOI 10.1109/SBAC-PADW60351.2023.00018

mation from the user source code; (iii) proposed a metric that helps to measure the quality of the memory access pattern w.r.t NUMA-awareness. Some NUMA profiling tools are available in the literature like *NUMAGrind* [8] and *NUMAPerf* [8], but they do not act at the source code level although considering other runtime aspects.

The rest of the paper is organized as follows. The next section provides the necessary background on NUMA architectures together with the frameworks we have used. Section III fully describes our profiling methodology, followed by our proposed metric for NUMA locality in Section IV. Illustrative and experimental results are presented in Section V. Section VI concludes the paper.

## II. Background

### A. Non-Uniform Memory Access (NUMA) architectures

Considering shared memory parallelization, the case of NUMA processors require special attention because of the particular organization of the memory and the impact on the overall performance. Figure 1 displays some NUMA configurations, illustrating the *non-conventional* sharing of the overall main memory. The NUMA configuration was

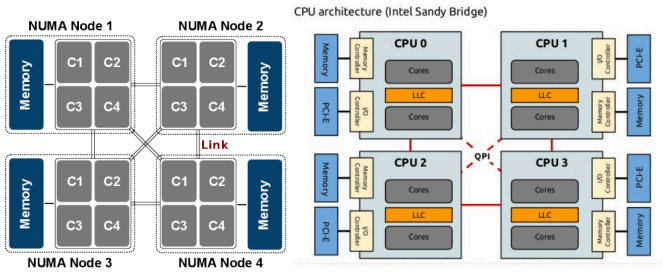


Fig. 1. Examples of NUMA configuration with four nodes

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, perfect scalability can be obtained only if all memory accesses are local. Indeed, remote accesses are more costly because of the additional mechanism that is activated to convey the data in addition to the contention on the affected QPI links (local accesses might be carried on simultaneously on these NUMA nodes). Skillful memory allocation and thread management for better scalability on NUMA processors is a hot topic. Stefan et al. [9] proposed a library for parallel programs on NUMA processors based on array abstraction and memory allocation routines. Their library allows automatic tuning of data placement and accesses for better scalability. Several specific contributions [10], [11], [12], [13], [14] investigate optimizing *threads and data placement* in a NUMA system by combining *data locality* and *thread binding* to reduce the occurrences of remote accesses. Lin et al. [15] proposed an efficient deployment of stencil computations on NUMA many-cores, targeting higher performance and portability.

### B. Flex: A lexical analyzer generator

To accomplish our objective of analyzing the memory access patterns of a given input program, we need to instrument its source code so as to produce the trace corresponding to its memory access pattern. To perform this source-to-source compilation task, we used *Flex* [16], a well-known lexical analyzers (scanners or lexers) generator. To proceed with Flex, the user defines a set of lexical patterns (*tokens*) and a set of actions to be performed whenever a token is encountered (so-called *rules*) during the parsing. All these specifications are written into a text file (*lex file*) that will be processed by Flex to produce the lexical analyzer in the form of a valid C file (called *lex.yy.c*) containing a function (called *yylex()*) that corresponds to the active parser itself. The compilation of this C source-code generated by Flex produces the user source-to-source "compiler" [17] that acts on any input text file according to *rules* specified at the design step. Figure 2 gives an overview of the steps when using Flex.

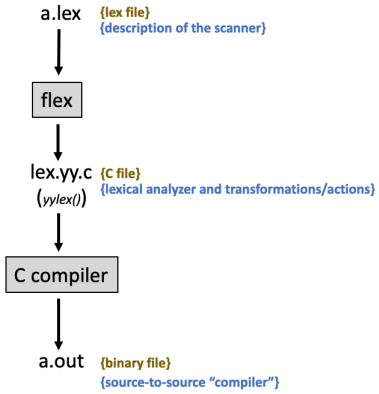


Fig. 2. Flex working diagram

### C. NUMA-related utilities: numactl and libnuma

As we have previously explained, our approach is to instrument the code so as to collect NUMA-related information about memory accesses. To be able to get this information, we need specific NUMA utilities that can provide them both for running threads and memory addresses. We considered *numactl* and *libnuma*.

*numactl* is a Linux utility that provides control over NUMA policy w.r.t to processes (threads) and memory. Indeed, *numactl* runs processes with a specific NUMA scheduling or memory placement policy. The policy is set for command and inherited by all of its children. In addition, it can set a persistent policy for shared memory segments or files. Since *numactl* is aware of the processor topology and how the CPU cores map to CPU sockets, it can also be used to get NUMA-related hardware information.

The *libnuma* is a library that offers a simple programming interface to the NUMA (Non-Uniform Memory Access) policy supported by the Linux kernel. *libnuma* allows the programmer, within its program, to get information like the NUMA node location of a given memory address as well as for a

given running thread and to dynamically perform specific (memory/thread) binding actions. For our work, we are mainly interested in getting NUMA-related information in order to generate our profile report as we are going to describe.

### D. Illustration of the NUMA impact

As previously explained, the impact of NUMA on efficiency mainly comes from *remote accesses* and *bus contention*. This impact is either an overhead of the overall execution time or bad scalability across several NUMA nodes. In the following example, we ran a single-threaded program (wave simulation code) on a machine with 8 NUMA nodes, using *numactl* to bind thread execution and memory allocation on a single specific node for each. Table I displays the matrix of the timings (the value at position $ij$ corresponds to the scenario where the program running on node $i$ and memory is allocated on node $j$). We can observe that this matrix has the same structure than the one related to the NUMA topology of the processor (obtained with `numactl --hardware`).

TABLE I
OVERHEAD OF REMOTE MEMORY ACCESSES

| | node 0 | node 1 | node 2 | node 3 | node 4 | node 5 | node 6 | node 7 |
|---|---|---|---|---|---|---|---|---|
| node 0 | 44.82 | 54.92 | 54.31 | 53.64 | 74.47 | 74.45 | 68.75 | 74.50 |
| node 1 | 54.08 | 44.63 | 53.58 | 53.24 | 73.72 | 73.45 | 74.53 | 68.64 |
| node 2 | 54.29 | 53.94 | 44.97 | 54.32 | 69.11 | 72.90 | 72.62 | 72.19 |
| node 3 | 53.97 | 54.06 | 54.65 | 44.44 | 73.03 | 69.06 | 72.49 | 72.60 |
| node 4 | 74.43 | 74.17 | 68.45 | 74.74 | 44.83 | 54.29 | 54.18 | 54.20 |
| node 5 | 75.45 | 74.01 | 74.57 | 68.99 | 54.04 | 44.49 | 54.05 | 53.98 |
| node 6 | 69.16 | 72.58 | 73.01 | 72.88 | 53.66 | 53.84 | 44.71 | 54.62 |
| node 7 | 73.08 | 68.52 | 72.94 | 72.49 | 53.72 | 53.68 | 54.56 | 44.60 |

A more complex example that illustrates the impact of NUMA effects on parallel scalability is taken from a work by Tadonki [2], where he proposes a NUMA-aware parallel implementation of Lattice Quantum ChromoDynamics simulation. Figure 3 displays the results of a NUMA-unaware version of the LQCD simulation code on a machine with 4 NUMA nodes. We can see the poor speedup when using several nodes.

| #cores | #threads | t(s) | GFlops | Speedup |
|---|---|---|---|---|
| 1 | 2 | 0.02552 | 9.98 | 1 |
| 2 | 4 | 0.01301 | 19.59 | 1.96 |
| 4 | 8 | 0.00679 | 37.50 | 3.76 |
| 8 | 16 | 0.00475 | 53.60 | 5.37 |
| (2 nodes) 16 | 32 | 0.00476 | 53.53 | 5.36 |
| (4 nodes) 32 | 64 | 0.00507 | 50.25 | 5.03 |

Fig. 3. Weak scalability of a NUMA-unaware parallel code

## III. OUR PROFILING METHODOLOGY

### A. Description of our profiling method

Our source code parser is primarily intended to help the programmer to profile his program with respect to memory accesses from the NUMA configuration standpoint. Considering an array, namely $A$, the idea is to replace each access $A[f(I)]$ by $g(cpu\_numa\_node, A, f(I))$, where $I$ is the iteration index and $g$ is a function that returns $A[f(I)]$ while extracting and recording profiling information considering that the access has been requested from NUMA node $cpu\_numa\_node$. The generated profiling information will be the triplet $(f(I), cpu\_numa\_node, mem\_numa\_node)$, where $mem\_numa\_node$ is the NUMA node *id* of the memory address of $A[f(I)]$. We always profile for **one array at a time** although our code transformation can simultaneously consider several arrays.

With *libnuma* library,

- $cpu\_numa\_node$ is obtained with the instruction `numa_node_of_cpu(sched_getcpu())` where `sched_getcpu()` returns the *id* of the executing CPU-core and `numa_node_of_cpu(cpu_id)` returns the NUMA node *id* of the CPU-core `cpu_id`.
- $mem\_numa\_node$ is obtained with the instruction `get_mempolicy(&mem_numa_node, NULL, 0, addr,MPOL_F_NODE | MPOL_F_ADDR)` where `addr` is the memory address of $A[f(I)]$.

Let us illustrate the transformation considering an array A. The instruction below

```
X = 3 + 2*A[3*i+j];
```

will be replaced by

```
X = 3 + 2*func(numa_node_of_cpu(
                sched_getcpu()),A,3*i+j);
```

At the end of the execution of the instrumented code, our profiling measurements are written down into a text file that we post-process in order to get a more compact form with associated statistics. Listing 1 provides an example of instruction with an array access and Listing 2 shows the corresponding transformation. Listing 3 describes how we get and save the profiling information.

```
x = A[B];
```

Listing 1. Input example

```
x = func(numa_node_of_cpu(sched_getcpu()), A,B);
```

Listing 2. Output example

```
void find_memory_node_for_addr(void* ptr) {
    int numa_node = 0;
    if(get_mempolicy(&numa_node, NULL, 0, ptr,
    MPOL_F_NODE | MPOL_F_ADDR) < 0)
        printf("WARNING: get_mempolicy failed    ");
    tab[2*pos+1] = numa_node;
}

int func(int caller,int x[], int y){
    tab[pos] = y;
    tab_caller[2*pos] = caller;
    find_memory_node_for_addr(x+y);
    pos++;
    if(pos == TAB_LENGTH){
        save();
        pos = 0;
    }
    return x[y];
}
```

Listing 3. Profiling fonctions

### B. Illustration of our profiling

To test our methodology and the associated tool, we consider the code for the simulation of seismic wave equation[1], focusing on NUMA locality. Our application kernel simulates the propagation of acoustic waves using a finite differences method with regular 3D grids. The wave equation uses second order time discretization and a second order spatial discretization, resulting in a 7-point stencil on a 3D domain. A more detailed description of our application and the numerical methods used is provided in [18]. Listing 4 is the code of our simulation kernel and Listing 5 is the corresponding instrumented version as generated by our tool following our instrumentation approach.

```
for(size_t n = 0; n < iterations; n++) {
  for(size_t i = STENCIL_RADIUS;
        i < nz - STENCIL_RADIUS; i++) {
    for(size_t j = STENCIL_RADIUS; j < nx -
    STENCIL_RADIUS; j++) {
      for(size_t k = STENCIL_RADIUS; k < ny -
    STENCIL_RADIUS; k++) {
         size_t current = (i * nx + j) * ny + k;
         double value = coefficient[0] *
            (prev_u[current]/dzSquared +
            prev_u[current]/dxSquared +
            prev_u[current]/dySquared);
         for(size_t ir=1; ir<=STENCIL_RADIUS; ir++){
           value += coefficient[ir] * (
            ((prev_u[current + ir] +
             prev_u[current-ir]) / dySquared) +
            ((prev_u[current + (ir * ny)] +
            prev_u[current-(ir * ny)]) /dxSquared)+
            ((prev_u[current+(ir*nx*ny)] +
            prev_u[current-(ir*nx*ny)])/dzSquared));
         }
         value *= dtSquared * vel_model[current] *
                vel_model[current];
         next_u[current] = 2.0 * prev_u[current] -
                 next_u[current] + value;
      }
    }
  }
```

Listing 4. Original version of our simulation kernel

```
for(size_t ir = 1; ir <= STENCIL_RADIUS; ir++){
 value += coefficient[ir] * (
 ((func(numa_node_of_cpu(sched_getcpu()), prev_u,
             current+ir) +
   func(numa_node_of_cpu(sched_getcpu()), prev_u,
            current-ir)) / dySquared ) +
 ((func(numa_node_of_cpu(sched_getcpu()), prev_u,
            current+(ir*ny)) +
   func(numa_node_of_cpu(sched_getcpu()), prev_u,
            current-(ir*ny))) / dxSquared ) +
 ((func(numa_node_of_cpu(sched_getcpu()), prev_u,
                  current+(ir*nx*ny))+
   func(numa_node_of_cpu(sched_getcpu()), prev_u,
             current-(ir*nx*ny)))/ dzSquared ));
}
value *= dtSquared * vel_model[current] * vel_model[
    current];
next_u[current] = 2.0 * func(numa_node_of_cpu(
    sched_getcpu()), prev_u,current) - next_u[
    current] + value;
```

Listing 5. Instrumented version of our simulation kernel

[1]https://github.com/HPCSys-Lab/wave-equation

After compiling and running the instrumented code, all collected profiling records are written down into a text file that contains one line of the form

`offset cpu_node mem_node`

for each access to the array of interest. The profiling report could be as follows

```
12  0  1
18  0  0
80  0  1
80  1  1
92  1  1
98  1  0
```

So, we can see that

- `prev_u[12]` was accessed from node 0 while being on node 1 (remote access)
- `prev_u[92]` was accessed from node 1 while being on node 1 (local access)
- `prev_u[80]` was accessed from node 0 and also from node 1 while being on node 1 (1 remote access and 1 local access)

From the basic form of the profiling report, we generate different other outputs by: *sorting* the lines by the `offset`; *aggregating* the accesses by the `offset` and adding the number of accesses, which yields lines of the form

`offset cpu_node mem_node, nb_accesses`

The compact form of the profiling report looks like Table II.

TABLE II
COMPACT PROFILING REPORT

| Offset | CPU node | MEM node | #Accesses |
|---|---|---|---|
| 13 | 1 | 1 | 6 |
| 13 | 0 | 1 | 4 |
| ... | ... | ... | ... |
| 162 | 1 | 1 | 30 |
| 162 | 0 | 1 | 20 |
| ... | ... | ... | ... |
| 332 | 1 | 0 | 70 |
| ... | ... | ... | ... |
| 1714 | 1 | 0 | 6 |

Note that we handle the profiling information through a global variable that is updated upon each access of the profile array. A mutual exclusion is needed in order to avoid race conditions due to several threads attempting an update at the same time, some profiling lines might be lost (overwritten).

The ultimate form of our profiling report is obtained from that of Table II by aggregating the lines by the pair `(cpu_node,mem_node)`. This gives a table where lines are of the form

`cpu_node mem_node, total_nb_accesses`

Table 4 displays an illustrative example from our experimental run.

Table III provides a quantitative overview of memory accesses from the NUMA standpoint. Its 2D matrix form is given by Figure 4 where the values are percentages.

TABLE III
PROFILING REPORT MATRIX

| CPU node | MEM node | Quantity |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 39106 |
| 1 | 1 | 30894 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 11.77 | 0 | 0 | 0 | 0.73 | 0 | 0 | 0 |
| 1 | 0 | 10.27 | 0 | 0 | 0.41 | 1.83 | 0 | 0 |
| 2 | 0 | 0 | 7.41 | 0 | 0 | 2.76 | 2.32 | 0 |
| 3 | 0 | 0 | 0 | 10.54 | 0 | 0 | 1.82 | 0.14 |
| 4 | 0.4 | 1.74 | 0 | 0 | 10.36 | 0 | 0 | 0 |
| 5 | 0 | 0.36 | 0.08 | 0 | 0 | 12.06 | 0 | 0 |
| 6 | 0 | 0 | 0.24 | 0.35 | 0 | 0 | 11.91 | 0 |
| 7 | 0 | 0 | 0 | 2.28 | 0 | 0 | 0 | 10.22 |

Fig. 4. Matrix of remote accesses

It can be used to appreciate the NUMA-awareness of the program under the guidance of a NUMA metric. We now describe a basic one.

## IV. A BASIC NUMA METRIC

Having a NUMA accounting in general is a helpful profiling item that the programmer can use to figure out the quality of memory accesses. However, for optimization purposes, having a single-value score (metric) is a goo complement as it can be considered as the objective function to be minimized. We define a basic one.

Consider the matrix corresponding to the profiling report like described in Figure 4, we denote $R = (r_{ij}), 1 \leq i, j, \leq N$, where $N$ is the number of NUMA nodes. The value of $r_{ij}$ represents the number of memory accesses from NUMA node $j$ to NUMA node $i$ (each of these requests comes from NUMA node $i$). Consider the NUMA distance matrix (given by the command `numactl --hardware`), we denote $D = (d_{ij})$. We define our NUMA locality metric as follows

$$\tilde{\delta} = \frac{1}{T \times Q} \sum_{1 \leq i,j \leq N} r_{ij} d_{ij}, \tag{1}$$

where $T$ is the total number of memory accesses ($T = \sum r_{ij}$) and $Q = \sum d_{ij}$. One could set the diagonal of $D$ to 0 (i.e. $d_{ii} = 0$, $1 \leq i \leq N$) in accordance with the consideration that there is no overhead with *local accesses* from the NUMA standpoint. In practice, these values are non-zero, are the same on the whole diagonal, and are lower than any other value. Thus, if these apply, then the aforementioned zeroing of the diagonal could be done by removing that constant to each of the entries of $D$ (i.e. replace $d_{ij}$ by $d_{ij} - d_{00}$).

**Very important points:** For future works or how to consider our approach (especially regarding the score), the two following key points are crucial.

- *Data locality.*
  In our approach, we count each memory access regardless of *cache line locality*. In practice, if the memory access pattern is (highly) regular, then each remote access will lead to local accesses for data within the same cache line (if done while that line is till in cache). Thus, counting each access individually severely evaluate the impact of NUMA effect. One way to do this from our fine grain profiling report is, for instance, to consider the cache line id and count for accesses at that granularity. This can be done by assuming an aligned allocation and considering both the length of the data type and that of the cache line. Some tools like *NumaPerf* deals with this at the level of memory pages.
- *Bus contention.*
  It is common to stress NUMA penalty because of the focus on *remote accesses* (they are indeed more costly than *local* ones). However, remote access might alleviate the contention on local memory buses and thereby improve performance and efficiency. Note that this was one of the motivation of the design of NUMA architectures. This is probably the reason why interleaving memory allocations statistically yields a good performance even in the presence of a significant amount of remote accesses.
- *Tracking all critical variables.*
  Our method allows to track a specific array. However, a given program might have several arrays that have a strong influence on the memory accesses efficiency. Thus, a more accurate profiling would be track all important arrays. This is a natural extension of our methodology.

## V. EXPERIMENTS

We consider our wave simulation code (the main loop) with four input parameters *dim_X, dim_Y, dim_Z, nb_iterations*. We influence memory allocation with either *membin* (allocate on a specific NUMA node), *–localalloc* (allocate on the NUMA node where the thread is running), and *–interleave=all* (allocate in a distributed way across NUMA nodes). Our goal is just to have different memory allocations and check the output of our profiling. We managed to have a balanced binding of the threads to the NUMA nodes.

We first consider the scenario (128, 128, 20) on an AMD EPYC 7402 processor (2 sockets with one NUMA node each, 24 cores per socket). Table IV summarizes our results on this processor. We can see the expected correlation between the execution time ($T$) and our metric ($\delta$).

TABLE IV
EXPERIMENTAL RESULTS ON A 2 NUMA NODES PROCESSOR

| N$^o$ | *scenario* | #threads | *mem policy* | $T$ | $\delta$ |
|---|---|---|---|---|---|
| 1 | $32^3 : 100$ | 2 | –localalloc | 0.002145 | 0.010499 |
| 2 | $32^3 : 100$ | 2 | –interleave=all | 0.002265 | 0.25000 |
| 3 | $64^3 : 100$ | 2 | –localalloc | 0.008956 | 0.126012 |
| 4 | $64^3 : 100$ | 2 | –interleave=all | 0.009263 | 0.2500025 |
| 5 | $128^3 : 20$ | 16 | –localalloc | 0.014295 | 0.017853 |
| 6 | $128^3 : 20$ | 16 | –interleave=all | 0.014714 | 0.253441 |

Now we consider another processor. An AMD EPYC 7601 32-Core Processor (2 sockets with 4 NUMA nodes each, 16 physical cores per NUMA nodes). The graph of the relative

distances between NUMA nodes is displayed in Figure 5 and our experimental results in Table V. About the aforementioned correlation between the execution time and our NUMA metric, which also shows up in Table V, the idea is to check that and increase (resp. decrease) in one applies to the other, thereby giving the programmer a quantitative objective that will guide his optimization efforts.

```
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  16  16  16  28  28  22  28
  1:  16  10  16  16  28  28  28  22
  2:  16  16  10  16  22  28  28  28
  3:  16  16  16  10  28  22  28  28
  4:  28  28  22  28  10  16  16  16
  5:  28  28  28  22  16  10  16  16
  6:  22  28  28  28  16  16  10  16
  7:  28  22  28  28  16  16  16  10
```

Fig. 5. Matrix of the distances between NUMA-nodes

TABLE V
EXPERIMENTAL RESULTS ON A 8 NUMA NODES PROCESSOR

| N$^o$ | *scenario* | #threads | *mem policy* | $T$ | $\delta$ |
|---|---|---|---|---|---|
| 1 | $32^3 : 20$ | 2 | all local | 0.000951 | 0 |
| 2 | $32^3 : 20$ | 2 | –localalloc | 0.001286 | 0.072029 |
| 3 | $32^3 : 20$ | 2 | –interleave=all | 0.001344 | 0.168942 |
| 1 | $64^3 : 20$ | 2 | all local | 0.011839 | 0 |
| 2 | $64^3 : 20$ | 2 | –localalloc | 0.006324 | 0.014311 |
| 3 | $64^3 : 20$ | 2 | –interleave=all | 0.006534 | 0.015693 |

## VI. CONCLUSION

We have proposed and implemented a profiling method that allows the programmer to get a trace of the memory accesses from the NUMA standpoint. Based on this profile report together with the NUMA distance matrix, we have built a metric that provides an indication about the quality of the memory access pattern w.r.t the NUMA characteristic. Like with any other kind of program optimization, having relevant profiling information is important and can be used as a key for a systematic approach. Indeed, using our profiling method (through the associated tool we have implemented[2]), the programmer gets an insight of the *NUMA locality* of his implementation, which he can then try to optimize under the guidance of our metric. This further step is left as future work together with other goals like: *extending our lexical analyzer to multidimensional access pattern*, *tracking multiple variables at once*, *refining the account of memory accesses considering locality w.r.t cache memory*, *generating suggestions for better memory allocation*.

## ACKNOWLEDGMENT

Thanks to *FAPESP* São Paulo Research Foundation (FAPESP) through grants 2022/11070-9 and 2022/00434-0 whose part of it has covered the visit of *Leticia Machado* at *Mines Paris-PSL* and thus initiated this work and the underlying cooperation. H.S. also thanks FAPESP for their support through grants 2019/26702-8 and 2023/00566-6.

[2]https://github.com/HPCSys-Lab/NUMA-metric

## REFERENCES

[1] J. Virieux and S. Operto, "An overview of full-waveform inversion in exploration geophysics," *Geophysics*, vol. 74, no. 6, pp. WCC1–WCC26, 2009. [Online]. Available: http://library.seg.org/doi/10.1190/1.3238367

[2] C. Tadonki, "Scalable numa-aware wilson-dirac on supercomputers," in *2017 International Conference on High Performance Computing & Simulation, HPCS 2017, Genoa, Italy, July 17-21, 2017*. IEEE, 2017, pp. 315–324. [Online]. Available: https://doi.org/10.1109/HPCS.2017.56

[3] O. Haggui, C. Tadonki, L. Lacassagne, F. Sayadi, and B. Ouni, "Harris corner detection on a numa manycore," *Future Generation Computer Systems*, vol. 88, pp. 442–452, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X1732188X

[4] B. Lepers, V. Quéma, and A. Fedorova, "Thread and memory placement on {NUMA} systems: Asymmetry matters," in *2015 USENIX annual technical conference (USENIX ATC 15)*, 2015, pp. 277–289.

[5] C. Tadonki, "High performance computing as a combination of machines and methods and programming," Ph.D. dissertation, Université Paris Sud-Paris XI, 2013.

[6] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. M. Lohman, "Numa-aware algorithms: the case of data shuffling." in *CIDR*, 2013.

[7] R. Al-Omairy, G. Miranda, H. Ltaief, R. M. Badia, X. Martorell, J. Labarta, and D. Keyes, "Dense matrix computations on numa architectures with distance-aware work stealing," *Supercomputing Frontiers and Innovations*, vol. 2, no. 1, pp. 49–72, 2015.

[8] X. Liu and J. M. Mellor-Crummey, "A tool to analyze the performance of multithreaded programs on numa architectures," in *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:4869589

[9] S. Kaestle, R. Achermann, and T. Roscoe, "Shoal: smart allocation and replication of memory for parallel programs," in *Proceedings USENIX Annual Technical Conference*, Santa Clara, USA, 2015, p. 810.

[10] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: A holistic approach to memory placement on numa systems," in *ASPLOS1*, Houston, Texas, USA, 2013, p. 810.

[11] B. Lepers, V. Quéma, and A. Fedorova, "Thread and memory placement on numa systems: asymmetry matters," in *In Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC 15)*, Berkeley, CA, USA, 2015.

[12] R. Lachaize, B. Lepers, and V. Quéma, "MemProf:A memory Profiler for NUMA Multicore Systems," in *USENIX ATC 12*, 2012.

[13] A. Collins, T. Harris, M. Cole, and C. Fensch, "LIRA: Adaptive Contention-Aware Thread Placement for Parallel Runtime Systems," in *ROSS*, 2015.

[14] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. Lohman, "LIRA: Adaptive Contention-Aware Thread Placement for Parallel Runtime Systems," in *http://www.pandis.net/resources/cidr13numashuffling.pdf*, 2013.

[15] P. Lin, Q. Yi, D. Quinlan, C. Liao, and Y. Yan, "Automatically Optimizing Stencil Computations on Many-core NUMA Architectures," in *International Workshop on Languages and Compilers for Parallel Computing*, Rochester, NY, USA, 2016.

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

[17] V. Paxson, W. Estes, and J. Millaway, "Lexical analysis with flex," *University of California*, p. 28, 2007.

[18] J. Freire de Souza, J. Baptista Dias Moreira, K. J. Roberts, R. di Ramos Alves Gaioso, E. Satoshi Gomi, E. C. Nelli Silva, and H. Senger, "`simwave` – A Finite Difference Simulator for Acoustic Waves Propagation," *arXiv e-prints*, p. arXiv:2201.05278, Jan. 2022.