Node-Level Performance Engineering and OpenMP

Christie Louis Alappat

November 30, 2015

Contents

1 Introduction **2**

2 Motivation **2**

3 Core Level **2**
3.1 Pipelining . . . 2
3.1.1 Some good practices . . . 3
3.2 SIMD . . . 3
3.2.1 Some Good practices . . . 4
3.3 Cache aware optimizations . . . 4
3.3.1 Cacheline and Prefetching . . . 5
3.3.2 Some optimisation techniques . . . 5

4 OpenMP **6**

5 Node Level **7**
5.1 ccNUMA . . . 8
5.2 Cache coherency . . . 8
5.3 NUMA aware programming techniques . . . 8

6 APPENDIX **10**
6.1 OpenMP Vector Triad . . . 10
6.2 NUMA Aware programming . . . 11

1 Introduction

Today simulations are getting more and more expensive and the never ending greed for faster codes are increasing. It is said that in the world of high performance computing only scalable codes will win. There are many questions one could face when confronting with code performance issues like: Why is my code not scaling? Why is my code not performing? What is my bottleneck? What is the limit of mt code? Is my code getting the optimum performance? and the list continues. Here we investigate on some of this topics and provide solution for these. In this report we first investigate on some performance techniques at core level then we move on to OpenMP and finally discuss about some node level optimisations .

2 Motivation

One could always think why to care about a single node, when one could have access to thousands of them, I could just use some parallelization technique and run them on big clusters. But if one blindly starts off with some parallelization technique without knowing his codes characteristics on node level, in most cases one would easily end up in a code that does not scale at all or a code with much less efficiency. Furthermore the performance gain on one node is magnified when it is run on a cluster, it also allows to make the full utilization of the available resources. Another important topic related to this field that is gaining rapid momentum is the energy consideration and energy efficiency. It is said that in future the code performance would not only be measured with MFLOPs or runtime but also Energy consumed.

3 Core Level

3.1 Pipelining

Pipelining is a strategy used by modern processors, to reduce the cycle times. Here a complex instruction is splitted into several simple but fast steps, each with same amount of time. This also allows for execution of different instructions simultaneously at different stages.Figure 1 [15] shows a multiplication pipeline in detail

The benefit of this complex strategy is that one result is obtained at each cycle after the pipeline is full. But another consequence of such a mechanism is one could easily loose a factor of 2-5 if pipelining does not happen correctly. We consider a simple example to make the fact more clear:

Consider a simple code that adds elements of an array

```
sum = 0
for ( i =0; i < N ;++ i )
sum = sum + A [ i ];
```

Here the variable sum is a dependency; since for every next iteration we need the value of sum computed, so this hinders all other computation and only one stage in add pipeline

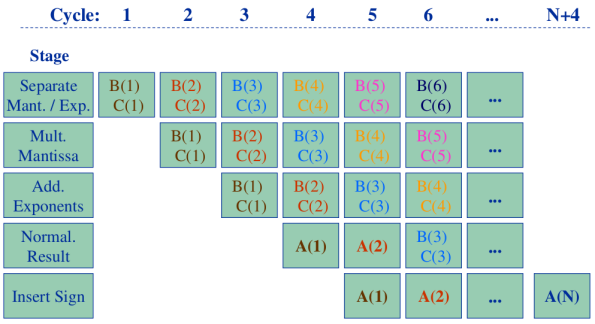

Figure 1: Multiplication Pipeline

is filled. In most modern cpu's we lose a factor of 3-5 since add pipeline consists mostly of 3-5 stages, and we can only fill one stage. So how could we do better,

Now consider the following snippet:

```
sum = 0
for ( i =0; i < N ; i = i +3)
{
sum1 = sum1 + A [ i ];
sum2 = sum2 + A [ i +1];
sum3 = sum3 + A [ i +2];
}
sum = sum1 + sum2 + sum3 ;
```

This is called loop unrolling here we have 3 sum variables, this implies if our add pipeline was 3 stage long we can fill our pipeline, since by the time sum3 is computed sum1 is already available to the next i. This would again make our pipeline's every stages busy. It is to note that one should handle the reminder loop if `N` is not a multiple of 3 (in this case).

3.1.1 Some good practices

Modern compiler does pipelining in some cases, if programmer allows it to. But most of the time pipelining of codes like above with dependency is not carried out. There can also be hidden dependency for eg: aliasing in `C++` , which should also be taken care (here one could provide proper compiler flags on use the keyword `restrict`) . If possible avoid branches in your routine. Also inlining subroutines might help pipelining. Also avoiding expensive operations like divide, sqrt,modulo etc. is definitely a good idea, since they have long pipeline stages and very high pipeline latency.

3.2 SIMD

SIMD stands for Single Instruction Multiple Data, It allows for concurrent execution of the same operation on wide registers. This may also be referred as code vectorization. Its a form of data level parallelism at core level. Most of the modern computers operate

on 256-bit AVX registers (i.e are capable of doing 4 operations in parallel). This factor is gaining more and more importance currently Xeon Phi processors have 512-bit registers, which implies if one fails to vectorize the code, he loses a factor of 8 immediately. The concept of SIMD can be understood better from the figure 2 [16]

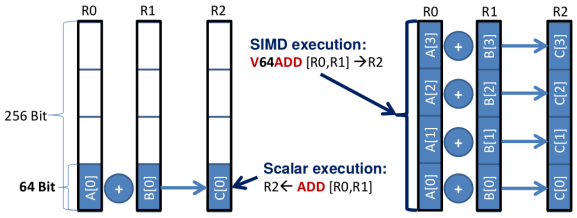

Figure 2: SIMD

Most of the factors that hinders Pipelining as mentioned in the above section 3.1 also hinders SIMD. Nowadays most of the modern compilers tries to do auto-vectorisation [2] whenever possible but there is no guarantee of the result. Its programmer's task to find whether the code is vectorised and if not to take appropriate action.

3.2.1 Some Good practices

- Ensure your code has a contiguous memory access since vectorisation can take place only if your code access contiguous memory. If memory is not contiguous then check whether restructuring data structure or accesses is possible.
- Ensure loops are of countable size and has one entry and exit point
- Check if the data is alligned.
- All the strategies in mentioned for pipelining 3.1.1 might also help vectorisation.
- Give useful hints to compiler to vectorise the code for eg. providing the compiler flag -march=native to produce specific code for the system's CPU [4]. Also indicating the vectorisable regions with pragmas like `#pragma omp simd`(for gcc) and `#pragma vector always` (for intel), might enable vectorisation.
- It is always a good practise to check whether compiler has vectorised your loop. Checking assembler code for packed instructions using `-S` flag, using `--ftree-vectorizer` [1] or using tools like `likwid` [7] might give you a detailed insight.

3.3 Cache aware optimizations

Nowadays data don not come directly from the memory but instead passes through several memory hierarchy called Caches to reach the processor. This is done inorder to reduce the latency of data transfer between memory and processors. Usually a modern CPU consists of 3 cache layer reffered to as L1, L2 and L3. L1 is the fastest(≈

$130 - 150GB/s$) but smallest($\approx 32KB$) and L3 is the biggest ($\approx 20 - 256MB$) but slowest($\approx 40GB/s$)[1] in the cache hierarchy, but it is to note that caches are significantly faster than Memory accesses.

Whether caches are beneficial or detrimental depends on how one use them and on the algorithm. If one accesses the data always from memory, without taking advantage of cache, the code might run slower than a computer where cache is absent, since reading and writing through different layers (hierarchy) costs clock cycles.

3.3.1 Cacheline and Prefetching

Furthermore to hide latency data transfers always happens at cache line granularity (typically 64 or 128 bytes). Also prefetching [2] is employed to further reduce the latency.

The above mentioned behavior of modern processors makes cache aware optimisations even more necessary, since if one is not aware of these behaviors then price to pay for it might be pretty high. For example if one does not pay attention to cache line concept and one accesses data randomly, then is data is sufficiently large he might experience a performance drop of a factor of 8, prefetching worsens the situation.

3.3.2 Some optimisation techniques

As opposed to the previous sections, where compilers were capable of applying *some* optimization techniques on its own, most of the compilers cannot employ any cache level or memory level optimizations on its own. So the responsibility is entirely on the programmer.Since most of the scientific codes are memory bound cache and memory optimisation might be a hotspot. Below are some techniques which could be employed:

- As previously mentioned due to cache lines and prefetching its always best to have a contiguous memory access, else we might end up not using the data brought in from the memory.
- Choose a cache friendly data structure.
- Blocking is a well known strategy to keep the data in cache, here one splits the data into chunks that would fit into cache and perform the maximum possible computation on them and then only brings in new data. But it is to note that excessive blocking might disturb vectorisation and prefetching.
- Know how the data is stored in memory: for eg `C++` stores in row-major order while `Fortran` in column-major order and access the data in the optimum format [11].
- Avoid leading dimensions of powers of 2 in arrays (for eg: 1024), since this might lead to cache thrashing. If necessary employ padding.

[1] Figures are just given to get an approximate idea

[2] Prefetching occurs when a processor requests an instruction from main memory before it is actually needed [10]

4 OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing [14]. It provides an incremental and easy way[15] to parallelise your code on Shared memory systems. The major advantage of OpenMP is its easiness since its build on pragma directives and some simple library functions. Although it doesn't offer fine grained parallelism, its sufficient for most of the scientific applications.The central concept of OpenMP are *threads*.[3] OpenMP uses a *fork - join* model of parallel execution. The concept could be easily understood from the Figure 3.

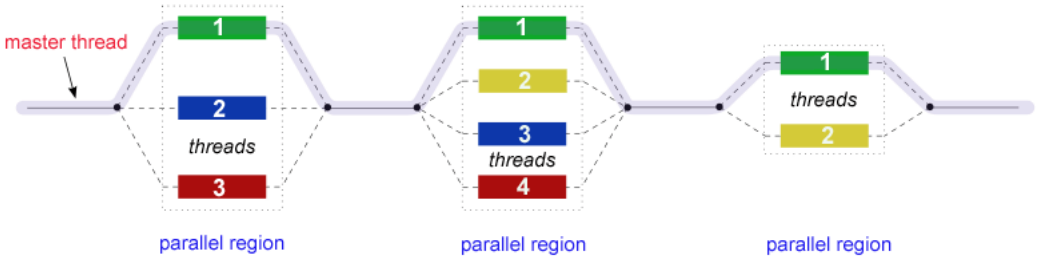

Figure 3: OpenMP - fork join model [13]

All OpenMP programs begin as a single process *the master thread*. The master thread executes sequentially until the first parallel region construct is encountered. At the parallel region master thread then creates a team of parallel threads, which work parallel to each other. When the parallel contruct is finished then they synchronise and then only master thread continues. In most of the case user do not need to worry about communication since all the threads work on a shared memory and data is visible to all, however one can also specify thread specific data using the `private` clause Because of

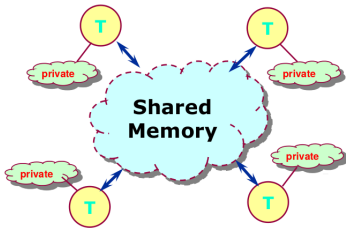

Figure 4: OpenMP - Memory Model [16]

this inherent shared memory one has to take care of avoiding race condition and deadlocks, which are common errors in OpenMP programs. An example of simple OpenMP Vector Triad could be found in appendix 6.1. The figure below5 shows the performance

[3] A thread of execution is the smallest unit of processing that can be scheduled by an operating system.

comparison on different threads when run on 2 sockets of Westmere LIMA cluster @2.20 GHz [8] with illustration of different regions of performance. From the graph one could

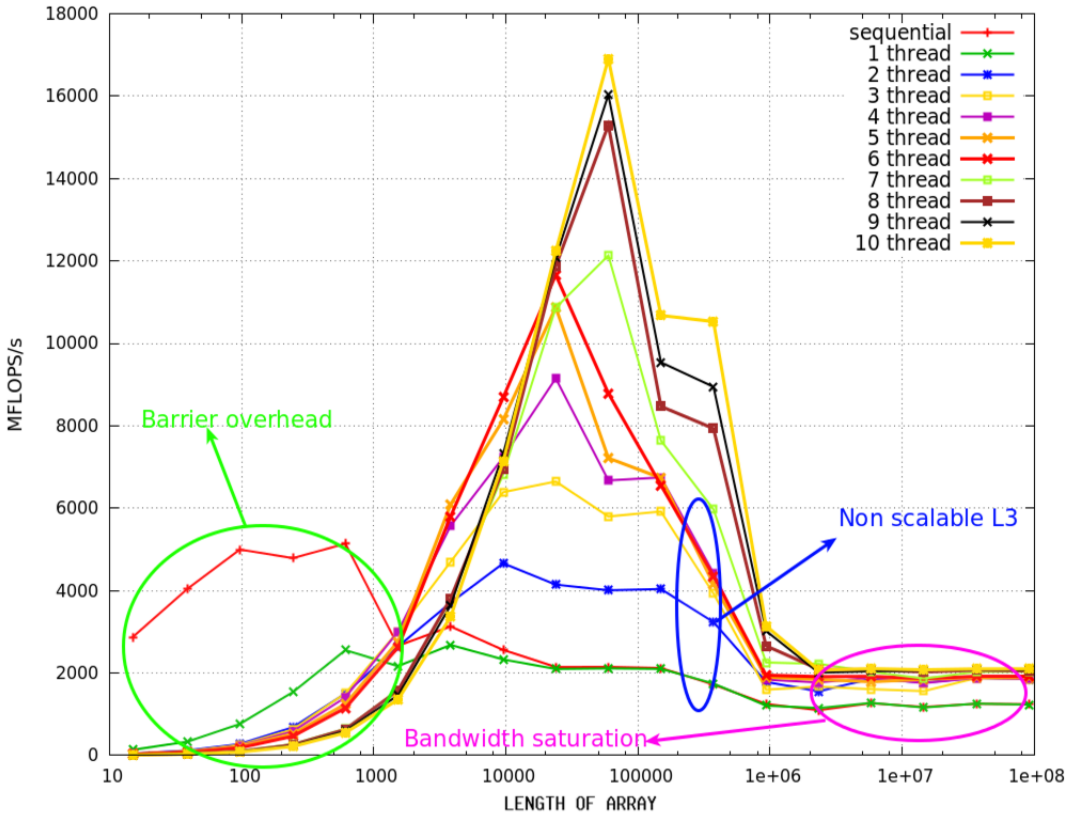

Figure 5: OpenMP - Vector Triad performance comparison

clearly see different overheads of OpenMP and the data sizes where such a parallelism would be beneficial in this simple case (note x-axis in logscale). It is to note that the non-scalable L3 behavior is due to an older architecture, this behavior would not be visible in modern architecture, albeit the scaling due to increased available bandwidth, when we shift from one socket to the next is visible from 6 thread to 7 thread transition at L3 regions.

5 Node Level

After discussing OpenMP we are ready to discuss on some performance tips on node level. Since OpenMP can be used for both UMA(Unified Memory Architecture) [12] and NUMA(Non Unified Memory Architecture) [9], it is a good candidate for our discussion. We will discuss here more on NUMA domains since modern cluster nodes are of these types and it also include UMA in its sub-domain see figure 6. Furthermore we look only at *Cache Coherent NUMA (ccNUMA)* systems.

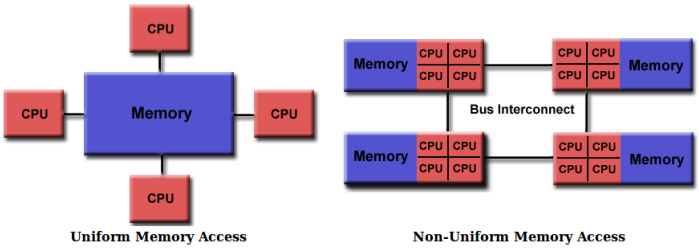

Figure 6: UMA vs NUMA architecture [13]

5.1 ccNUMA

ccNUMA is a shared memory architecture, here memory is physically distributed but all processors in the domain can access any memory (local or remote) since they are all connected to each other using links called HyperTransport (AMD)[5] or QuickPath Interconnect [6] (Intel). The important message to take away is that eventhough any processor can access data from any memory, bandwidth and latency is different for local and remote memories, and as one could predict local access is much faster than remote. Most of the programmer are not aware of such a heterogeneity in the system and if one does not take this into account it might lead to severe performance consequences (for eg: lack of scaling).

5.2 Cache coherency

Since all processors can access data from memory, leads to another problem namely the *consistency* of the data. Multiple copies of the same data is present in caches of different processor. In order to avoid the ambiguity and to have a consistent data view a cache coherency protocol is employed which ensures a consistent data view. As one could imagine this cannot be employed for free, but it has a substantial overheads, in terms of write back and invalidation of data.

5.3 NUMA aware programming techniques

As can be seen from above one could easily come to a conclusion that to use local memory as far as possible 7, which as dual benefits of increased bandwidth (less latency) and less of overhead due to cache coherence protocol. But now comes the question, where is my data ? This could be clearly understood by the following Golden Rule for ccNUMA[15] systems:

“ A memory page is mapped into the locality domain of the processor that first writes to it ”

The rule clearly states that the mapping happens at the time of initialization (i.e writing). It is to note that simply allocating memory is not sufficient to map the data. This also tells us it is always a good idea to do computation with the data which the specific thread initialized. Also pinning threads to processors are really important else

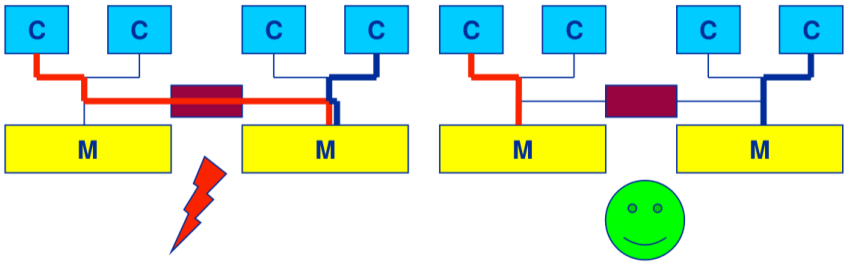

Figure 7: Good and Bad ccNUMA access [16]

they might migrate and work on remote data. Furthermore touching a single item is sufficient to map the entire page. An example of a NUMA aware programming could be found in Appendix 6.2. A comparison of running the code 6.2 vs a non- NUMA initialized code on 2 sockets of Ivy Bridge cluster- Xeon 2660v2 (Emmy cluster)[3] is shown in the figure 8 It can be seen the breakpoint between 2 graphs starts once we

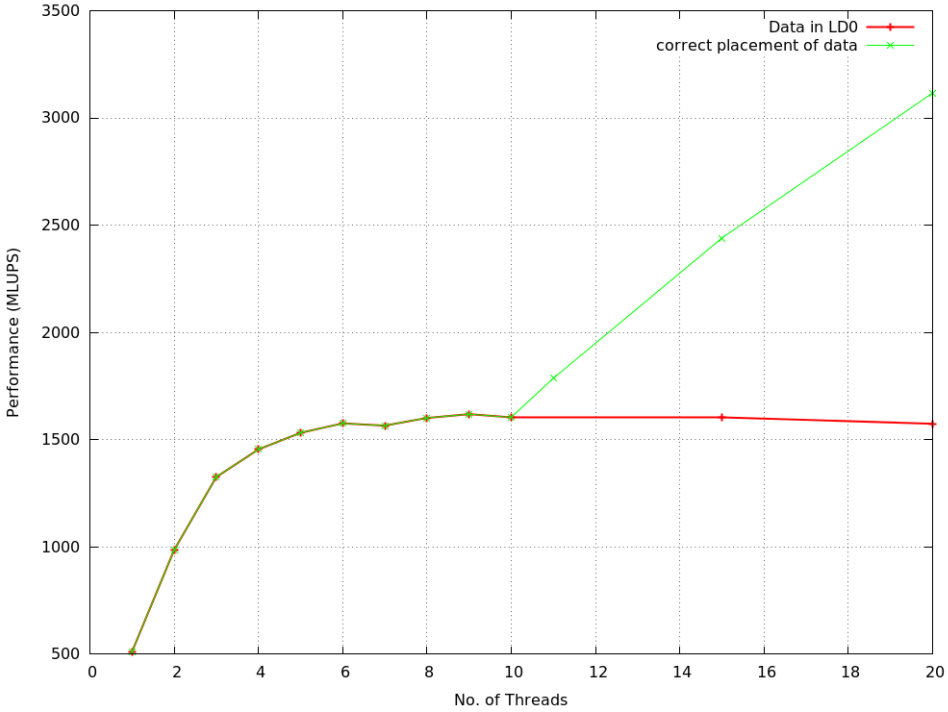

Figure 8: NUMA aware vs incorrect data placement

start using the second socket (here at 11 the thread). The poor scalability of incorrect placement is clearly seen from the graph.

6 APPENDIX

6.1 OpenMP Vector Triad

```
int main()
{
int num_thread;
num_thread=0;

#ifdef _OPENMP
#pragma omp parallel private(num_thread) //private clause for
                                         //thread private data
{
 num_thread=omp_get_num_threads(); //library call to get total
                                   //no.of threads
 int myid=omp_get_thread_num();   //library call to get thread id

 if(myid==0)
   std::cout<<num_thread<<"\n";
}

#else
std::cout<<num_thread<<"\n";
#endif

int n;
double *a,*b,*c,*d;
n=(pow(2.5,k));

//allocate data
a=new double[n];
b= new double[n];
c=new double[n];
d= new double[n];

//initialise data
#pragma omp parallel for schedule(rutime) //combined workshare
   construct, begin of parallel part and worksharing construct

 for(int i=0;i<n;++i)
 {
  a[i] = 3;
  b[i] = 4;
  c[i] = 5
  d[i] = 6;
  }

struct timeval tym;
gettimeofday(&tym,NULL);
double wcs=tym.tv_sec+(tym.tv_usec/1000000.0);

//benchmark loop
#pragma omp parallel  //begin of parallel part
{
 #pragma omp for schedule (runtime) //worksharing construct
 //parallel for splits the total iteration into chunks that are
 //computed by different threads
 for(int i=0;i<n;i++)
```

```
 {
  a[i]=b[i]+c[i]*d[i];
 }
}
gettimeofday(&tym,NULL);
double wce=tym.tv_sec+(tym.tv_usec/1000000.0);
runtime = wce-wcs;
std::cout<<n<<" \t"<<(2*n*(0.000001))/runtime<<std::endl;
delete[] a;
delete[] b;
delete[] c;
delete[] d;

return 0;
}
```

6.2 NUMA Aware programming

```
int main()
{
        const unsigned int Size=4000;
        int thread_num;
        double *phi0;
        double *phi1;

        //Allocation: Note mapping does not take place here
        phi0=new double[Size*Size];
        phi1=new double[Size*Size];

        #pragma omp parallel
        {
        //numa aware initialisation
       //This avoids serial initialisation;
       //schedule should be static inorder to make sure
       //each thread uses the data it initialised

                #pragma omp for schedule(static)
                for(int i=0;i<Size;++i)
                {
                        for(int j=0;j<Size;++j)
                        {
                                phi1[i*Size+j]=phi0[i*Size+j]=1;
                        }
                }


                //stencil
                #pragma omp for schedule (static)
                //assuming 0 on boundary
                for(int i=1;i<(Size-1);++i)
                {
                        for(int j=1;j<(Size-1);++j)
                        {
                                phi1[i*Size+j]=0.25*(phi0[(i+1)*
                                   Size+j]+phi0[(i-1)*Size+j]+
```

```
                                    phi0[i*Size+j+1] +phi0[i*Size+j
                                        -1]);
                            }
                    }
            }
            std::copy(phi1,&phi1[Size],phi0) //Interchange the arrays
            delete[] phi0;
            delete[] phi1;
            return 0;
}
```

References

[1] *Auto-vectorization with gcc.* https://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[2] *Auto-vectorization with gcc 4.7.* http://locklessinc.com/articles/vectorize/.

[3] *Emmy cluster.* https://www.rrze.fau.de/dienste/arbeiten-rechnen/hpc/systeme/emmy-cluster.shtml.

[4] *Gcc optimization.* https://wiki.gentoo.org/wiki/GCC_optimization.

[5] *Hypertransport.* https://en.wikipedia.org/wiki/HyperTransport.

[6] *Intel qpi.* https://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect.

[7] *Likwid.* https://github.com/RRZE-HPC/likwid.

[8] *Lima cluster.* https://www.rrze.fau.de/dienste/arbeiten-rechnen/hpc/systeme/lima-cluster.shtml.

[9] *Numa.* https://en.wikipedia.org/wiki/Non-uniform_memory_access.

[10] *Prefetching.* https://en.wikipedia.org/wiki/Instruction_prefetch.

[11] *Row-major order.* https://en.wikipedia.org/wiki/Row-major_order.

[12] *Uma.* https://en.wikipedia.org/wiki/Uniform_memory_access.

[13] B. Barney, *Openmp lawrence livermore national laboratory.* https://computing.llnl.gov/tutorials/openMP/.

[14] OpenMP, *Openmp.* http://openmp.org/wp/.

[15] G. Wellein and G. Hager, *Introduction to high performance computing for scientists and engineers.*

[16] G. Wellein, G. Hager, and M. Wittmann, *Programming technique for super computers*, 2015.