Digital Object Identifier

Vectorized Highly Parallel Density-based Clustering for Applications with Noise

JOSEPH ARNOLD XAVIER[1,2], JUAN PEDRO GUTIÉRREZ HERMOSILLO MURIEDAS[3], STEPAN NASSYR[1], ROCCO SEDONA[1] (Member, IEEE), MARKUS GÖTZ[3] (Member, IEEE), ACHIM STREIT[3], MORRIS RIEDEL[2] (Member, IEEE), GABRIELE CAVALLARO[1,2] (Senior Member, IEEE)

[1]Jülich Supercomputing Centre (JSC), Forschungszentrum Jülich, 52428 Jülich, Germany (e-mail: j.arnold@juelich.de, s.nassyr@fz-juelich.de, g.cavallaro@fz-juelich.de, r.sedona@fz-juelich.de)
[2]School of Engineering and Natural Sciences, University of Iceland, 107 Reykjavik, Iceland (e-mail: morris@hi.is)
[3]Scientific Computing Center (SCC), Karlsruhe Institute of Technology, 76344 Eggenstein-Leopoldshafen, Germany (e-mail: juan.muriedas@kit.edu, markus.goetz@kit.edu, achim.streit@kit.edu)

Corresponding author: Joseph Arnold Xavier (e-mail: j.arnold@fz-juelich.de).

This work was performed in the EUPEX (European Pilot for Exascale) project, which received funding from the European High-Performance Computing Joint Undertaking (EuroHPC JU) under grant agreement No 101033975. This work received complementary funding from the German Federal Ministry of Education and Research (No. 16HPC066K) and by the EUROCC2 project, funded by the EuroHPC JU and EU/EEA states under grant agreement No 101101903. Part of the experiments were performed on the HoreKa supercomputer, funded by the Ministry of Science, Research, and the Arts Baden-Württemberg and the Federal Ministry of Education and Research.

ABSTRACT Clustering in data mining involves grouping similar objects into categories based on their characteristics. As the volume of data continues to grow and advancements in high-performance computing evolve, a critical need has emerged for algorithms that can efficiently process these computations and exploit the various levels of parallelism offered by modern supercomputing systems. Exploiting Single Instruction Multiple Data (SIMD) instructions enhances parallelism at the instruction level and minimizes data movement within the memory hierarchy. To fully harness a processor's SIMD capabilities and achieve optimal performance, adapting algorithms for better compatibility with vector operations is necessary. In this paper, we introduce a vectorized implementation of the Density-based Clustering for Applications with Noise (DBSCAN) algorithm suitable for the execution on both shared and distributed memory systems. By leveraging SIMD, we enhance the performance of distance computations. Our proposed Vectorized HPDBSCAN (VHPDBSCAN) demonstrates a performance improvement of up to two times over the state-of-the-art parallel version, Highly Parallel DBSCAN (HPDBSCAN), on the ARM-based A64FX processor on two different datasets with varying dimensions. We have parallelized computations which are essential for the efficient workload distribution. This has significantly enhanced the performance on higher dimensional datasets. Additionally, we evaluate VHPDBSCAN's energy consumption on the A64FX and Intel Xeon processors. The results show that in both processors, due to the reduced runtime, the total energy consumption of the application is reduced by 50% on the A64FX Central Processing Unit (CPU) and by approximately 19% on the Intel Xeon 8368 CPU compared to HPDBSCAN.

INDEX TERMS High-performance computing, Density-based clustering, Vectorization, VHPDBSCAN.

I. INTRODUCTION

CLUSTERING algorithms are capable of discovering patterns in a dataset by maximizing a similarity measure between items within a group and minimizing it between different clusters. In the age of Big Data, manual labeling has become prohibitively expensive, making scalable and parallel cluster analysis highly sought after in multiple fields, e.g., satellite image segmentation [7], point cloud analysis [28], or customer data analysis [33].

Modern computing systems are capable of parallelizing software at multiple levels. At the lowest level, CPUs use instruction pipelining and super-scalarity (or instruction parallelism) to process multiple low-level instructions at a time in a single processing unit. SIMD instructions offer data parallelism by having the execution of the same instruction or operation on multiple units of data, consuming fewer clock cycles. In order to cater to the ever-increasing demand for vector processing capability by HPC workloads, most modern CPUs support SIMD instruction sets that can operate on extra-wide registers. For example, Intel currently provides up to 512-bit SIMD computations through their AVX (Advanced Vector Extensions) instructions. ARM introduced a vector length agnostic SIMD architecture model called Scalable Vector Extension (SVE) [29] to support up to 2048-bit vectors. SIMD instructions can be used to operate on different data types, ranging from integers to floating points of various precision. In order to exploit the SIMD capabilities of the processor, it is important that algorithms allow computations on multiple units of data simultaneously. In this paper, we present a vectorized implementation of HPDBSCAN called VHPDBSCAN that efficiently uses the SIMD instructions to accelerate distance computations, which form the most compute intensive part of the DBSCAN algorithm. Our key contributions in this work are:

- We introduce an optimized, vectorized algorithm referred to as VHPDBSCAN, which efficiently leverages SIMD instructions to accelerate the computationally intensive distance calculations required by the DBSCAN algorithm. We also parallelize the computation of neighbouring cells of a given cell in the grid, which is particularly useful in the analysis of high dimensional datasets.
- Implementations of this algorithm on an ARM based CPU – A64FX – providing SVE [29] intrinsics as well as for x86-based Intel CPU – Intel Xeon 8368 – offering AVX-512 SIMD operations.
- An evaluation of the performance and energy consumption at the core and node level in comparison with HPDBSCAN.

II. BACKGROUND

A. DENSITY-BASED CLUSTERING FOR APPLICATIONS WITH NOISE

One of the most popular clustering algorithms used today is DBSCAN [15]. The algorithm groups points that lie close to each other into clusters and identifies points that are not a part of any dense group as noise. DBSCAN is parameterized by two values: the maximum distance ϵ between two points for them to be considered as neighbours, and the minimum number of neighbouring points min_points that a point needs to have for it to be considered part of a dense region. DBSCAN recursively expands clusters by identifying dense regions. A point p is marked as a core point if it has at least min_points number of points within ϵ distance from it. A point q is said to be directly reachable from point p if q is within ϵ distance from p and p is a core point. A point q is density reachable from p if there is a path $(p_1, ..., p_n)$ with p_1 = p and p_n = q, where each point p_{i+1} is directly reachable from p_i. Note that in this case all points in $(p_1, ..., p_n)$ are core points, with the possible exception of q, as it may not have min_points neighbours. In such a case q is marked as a border point. Figure 1 illustrates DBSCAN clustering with $min\_points = 3$. The core points are marked in black, while the border point and the noise point are marked in grey and white respectively. Note that each core point contains at least three points including itself within ϵ distance. Also, the border point in the figure is connected to just one core point. Also in the bottom circle, the lowermost point is a border point. The general procedure of the queue-based variant of DBSCAN is described in Algorithm 1.

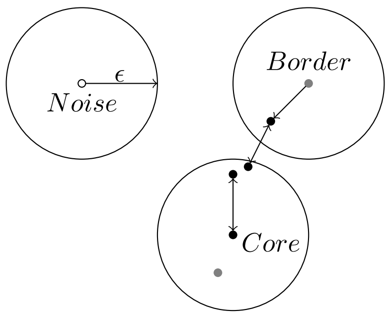


FIGURE 1: DBSCAN clustering with $min\_points = 3$

Algorithm 1: DBSCAN

```
Data: Dataset $X$, $\epsilon$, $min\_points$
Result: $clusters$, $noise\_points$
for $p$ in $X$ do
    $cluster_p = new\ cluster()$;
    if $!visited(p)$ then
        $compute = new\ queue()$;
        $compute.insert(p)$;
        while $length(compute) \neq 0$ do
            $i = compute.front()$;
            $compute.pop()$;
            $N = getNeighbours(i, X, \epsilon)$;
            if $length(N) \geq min\_points$ then
                for $c$ in $N$ do
                    if $!visited(c)$ then
                        $compute.push(c)$;
                        $cluster_p.add(c)$;
                        $visited(c) = true$;
                    end
                end
            else
                $markNoise(i)$;
            end
        end
    end
end
```

In Algorithm 1, the *getNeighbours* function computes the distances between point i and every point in X and returns a list of points from X that lie within ϵ distance from point i. The time complexity of the algorithm mainly lies in the number of distance computations necessary to determine if two points are neighbours. The original DBSCAN paper [15] shows that the time complexity of the algorithm can be reduced from $O(n^2)$ to $O(n \log(n))$ using R-trees.

B. HIGHLY PARALLEL DENSITY-BASED CLUSTERING FOR APPLICATIONS WITH NOISE

HPDBSCAN [18] introduces both shared memory parallelism and multi-core parallelism to the original DBSCAN algorithm. The implementation uses Message Passing Interface (MPI), together with Open Multi-Processing (OpenMP), to distribute the expensive distance computation of DBSCAN on multi-node and multi-core systems, making it ideal for a High Performance Computing (HPC) environment. HPDBSCAN uses an indexing structure to simplify the search of neighbouring points, but instead of using R-trees [5], it uses a grid-based indexing approach [19]. The stages of HPDBSCAN can be broadly divided into four major stages. In the first step, the entire dataset is split into equally sized chunks, based on the number of MPI ranks. Hence each MPI rank loads only a part of the original dataset. The d-dimensional bounding box containing the data is evenly split into d-dimensional cells with a side length of ϵ. Each of the d-dimensional data points is assigned to a unique spatial cell corresponding to their location within the data space. The data points are then sorted so that points close to each other are placed together in memory. A hash map is used to quickly map from a cell index to the initial location of its data points in memory and the number of points assigned to the cell. To balance the computational load for each worker, a cost heuristic calculates the number of comparisons between each point in a cell and its neighbouring cells. The number of neighbouring cells for a given cell in the grid can be up to 3^d where d is the number of dimensions of the dataset. Based on the computed cost, the subspaces are divided among the workers. This is particularly important in highly skewed datasets. Additionally, to ensure each worker has access to the relevant neighbouring cells, *halo* or *ghost* cells are replicated in workers with a shared border in the hypergrid to avoid remote memory accesses. The next step is the local computation of clusters carried out by each parallel worker, which we refer to as local DBSCAN, as it closely follows the original implementation of DBSCAN. Here, each worker iterates over the points within the cells that have been assigned to it. The number of neighbours is determined for each point based on the distance ϵ, to determine if it is equal or greater than min_points. It is also checked if it is part of the neighbourhood of a core point and is accordingly assigned a cluster label. At the end of this stage, every point has been assigned a cluster label (or determined to be noise) by the worker. This stage is the most compute-intensive stage of the clustering process, making it an ideal target for vectorization. We discuss this stage in detail in Section II-B. The final stage is the rule-based merging of clusters. Here, HPDBSCAN makes use of the labels of the halo cells to merge clusters that were computed by more than one parallel worker. The points in the *halo* cells will have conflicting labels between the different workers, and cluster labels of these points are used to merge parts of the same cluster that lay across multiple grid spaces. The resulting cluster labels are broadcast so that each node will directly map the local cluster label to a global one.

Local DBSCAN

As explained earlier, the most compute-intensive stage of HPDBSCAN is the computation of local clusters by each worker. Measurements of execution times have shown that the *local DBSCAN* function alone accounts for up to 80% to 90% of the total execution time. The rest of the time is spent computing the keys for each point, sorting the cells, and distributing the dataset to each worker. Algorithm 2 outlines the steps. *Local DBSCAN* function returns the local cluster labels computed for each point in the subspace. The cluster labels of the halo cells are used to determine the global cluster labels later in the merging stage. Each worker is equipped with a list of direct mappings from its locally computed subcluster label to a global label. A C++ `std::unordered_map` data structure called *Rules* is used to the mapping from a temporary cluster label to a global cluster label. The *getCellIndex* function returns the

Algorithm 2: Local DBSCAN

```
Data: Dataset $X$, $\epsilon$, $min\_points$
Result: Rules
$rules = Rules()$;
for $p$ in $X$ do
    $index = getCellIndex(p)$;
    $potential\_Nb = getNeighbours(index)$;
    if $length(potential\_Nb) \geq min\_points$ then
        $label, neighbours =$
          $regionQuery(potential\_Nb, min\_points, \epsilon)$;

        if $length(neighbours) \geq min\_points$ then
            $markAsCorePoint(p)$;
            for $q$ in $neighbours$ do
                if $isCorePoint(q)$ then
                    $markAsSame(rules, label[q], label)$;

                end
            end
        end
    else if $!visited(p)$ then
        $markAsNoise(p)$;
    end
end
```

index value of the current point in the sorted dataset and the *getNeighbours* function returns a list of points from the neighbouring cells of the current point p including those from the same cell as p. While *potential_Nb* contains the list of points from neighbouring cells, the *regionQuery* function computes the distance from each point with all the points in *potential_Nb* and identifies all the points within ϵ distance from point p. It also returns a temporary cluster label for each point. Later, the cluster labels of the core points are updated such that they share the same label. The iteration over all the points in the allocated subspace and the determination of the local cluster labels is done in parallel using shared memory parallelization by using OpenMP *parallel for* pragma in the outermost loop of the Algorithm 2. The OpenMP threads are scheduled dynamically with an emprically determined chunk size of 40 [18]. As the iterations of local DBSCAN function are parallelized using shared memory parallelism offered by OpenMP, overlapping of the ϵ neighbourhood from different threads can happen, leading to a data race condition. To avoid this, a simple atomic *min* operation is used to set the cluster label and the core property at once. Also, it is important to note that the distance computation between a particular point and its neighbours is not done in parallel and is done iteratively. The algorithm of *regionQuery* is given below.

```
Algorithm 3: regionQuery
  Data: Point $p$, Points $potential\_Nb$, $\epsilon$, $minPoints$
  Result: $label$, $neighbours_p$
  $label = INT\_MAX$;
  for $q$ in potential_Nb do
      if $euclideanDistance(p, q) \leq \epsilon$ then
          $add(neighbours_p, q)$;
          if $isCorePoint(q)$ then
              $label = min(label, getLabel(q))$;
  end
```

Even though the number of distance computations is reduced by the usage of the grid-based indexing structure, and multiple cells and points can be processed at the same time thanks to parallelization due to MPI and OpenMP, the runtime of HPDBSCAN is still largely determined by the distance computation between points across neighbouring grid cells. A possible solution to improve the performance of the individual workers computing the distance computation is to use the vectorization features available in modern CPUs. In the context of DBSCAN, SIMD instructions can be used to simultaneously compute the distance between a particular point and multiple points, and to determine which of these points is within the distance ϵ. This would result in each thread fully leveraging the SIMD capabilities of the processor and subsequently being allotted more points on each iteration of the local DBSCAN. We explain vectorization of the local DBSCAN algorithm in Section IV.

The rest of this work is divided in the following sections: Section III surveys the related work. A detailed explanation of our algorithm VHPDBSCAN and optimizations to the process of assigning cluster labels is provided in Section IV. A brief introduction to the SVE supported by ARM architecture based processors, and some technical details necessary to implement VHPDBSCAN on the A64FX processor are provided in Section IV-B. The experimental evaluation of VHPDBSCAN on three different datasets on the A64FX processor where we study the scaling behavior of our implementation is explained in Section V. In Section V-D, we demonstrate the performance improvements and the subsequent reduction in the CPU core and memory energy consumption using VHPDBSCAN over HPDBSCAN on two different processors: the A64FX using the SVE instruction set and the Intel Xeon Platinum 8368, which uses the AVX-512 instruction set.

III. RELATED WORK

Given their importance, a number of efforts to parallelize popular clustering algorithms have been attempted. Ali et al. [4] suggest a parallel variant of the K-means algorithm using multiple CPU cores of a single machine. Woodely et al. [32] developed a parallel variant of the K-tree algorithm that uses a tree-like data structure where clusters are represented as leaf nodes and the cluster representations are stored in non-leaf nodes. Hasib et al. [3] demonstrated that applying vectorization to the K-means algorithm yielded a performance improvement of 4.5x and a decrease in energy consumption by a factor of 8 on an Intel i7 Haswell machine. The earliest work on parallelizing the DBSCAN algorithm was Parallel DBSCAN (PDBSCAN) [34], and is based on a distributed R*-tree implementation. Here, the indices are built on a master node and the dataset is split and distributed to slave nodes, where local clustering is performed. Later, the local clusters are merged by re-clustering the bordering regions of the split. Coppola and Vanechi [13] present a queue-based DBSCAN implementation where each core point is processed one at a time, but the neighbours are computed in parallel and placed in the queue. Januzaj et al. [22] present a distributed algorithm for analysis of data residing on different systems in local or wide area networks (LANs/WANs). Here, density is computed on data located at a local site and according to a quality criterion, they are chosen to serve as local representatives to a server site. At the server site, they are clustered with density-based clustering algorithm. The determination of local representatives is done in parallel. Brecheisen et al. [9] show how simpler lower bounding distance functions can be used to efficiently parallelize DBSCAN. Parallel disjoint set based DBSCAN (PDSDBSCAN) [25] is a parallelized implementation of DBSCAN that uses the disjoint-set data structure and a tree-based bottom-up approach to construct the clusters. GridDBSCAN [24] uses a grid-based approach for spatial locality information in order to reduce the number of neighbourhood queries. The key drawback of the approaches stated above are that the indices are replicated on each of the slave workers and the pre-processing step has to be done entirely on the main memory in the master node, thus making it non scalable for large databases. μDBSCAN [27] proposes a scalable DBSCAN implementation without the computation of neighbourhood queries by using micro clusters for clustering extremely large datasets. The implementation is still slower than HPDBSCAN by a factor of 2 according to the paper but significantly faster than PDSDBSCAN and GridDBSCAN. Wang et al. [31] present highly parallel algorithms for exact computation of DBSCAN for 2D and higher dimensional datasets and also approximate computation of clusters. Unlike HPDBSCAN, the algorithm is not scalable at the node level. [2] suggest a parallel implementation of Statistical DBSCAN for execution on Spark based cluster on Google Cloud Platform. [26] suggest a block-based distance matrix computation on Intel Many core processors. However, we do not compute a distance matrix as such, but instead compute the distances between a point and its potential neighbours and update the cluster labels of all the points accordingly.

IV. VHPDBSCAN

A. VECTORIZATION OF LOCAL DBSCAN

The heart of the local DBSCAN algorithm lies in the distance computation between a point and its potential neighbours and is clearly the most compute-intensive operation in the clustering process. We now explain the strategy to vectorize the distance computation and label assignment in the *local DBSCAN* function. The pseudo-code to compute the Euclidean distance is given in Algorithm 4. Calculating the square root in the final step is deliberately omitted as, for comparison purposes, we can directly compare the squared values of both distances.

Algorithm 4: calculateEuclideanDistance

```
Data: Point $p$, Point $q$, $dimensions$
Result: $distance$
$distance$ = 0.0;
for $d = 0$ to $dimensions$ do
    $distance = distance + (p[d] - q[d])^2$;
end
```

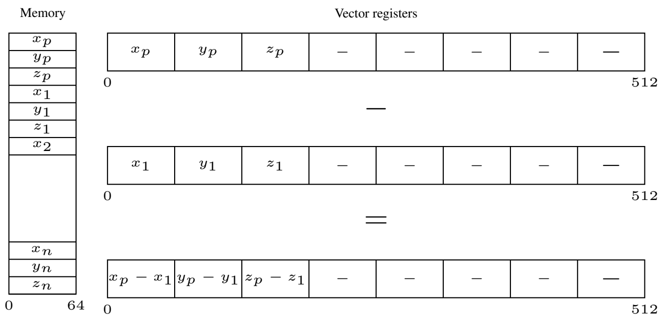


FIGURE 2: Vectorized computation of Euclidean distance in a three-dimensional dataset

A naïve approach to vectorize the distance computation illustrated in Algorithm 4 is by loading p and q into vector registers and applying SIMD subtraction and multiplication on the registers as shown in figure 2. The key drawback of this step is that the number of elements that can be loaded per iteration is limited by the dimensions of the data point. Note that the size of each vector register in Figure 2 is eight but only three values have been loaded into the register resulting in under-utilization of the vector registers. The datasets that we experimented with had dimensions that were mostly within five. Note that a SIMD register of width 512 bits can hold up to 16 double precision floating point values or 32 single precision floating point values. In the case of five-dimensional vectors, it would lead to using 160 out of the total 512 bits available in the vector register.

To exploit the larger width of the vector registers, we need to exploit the possibility of loading multiple data points onto the vector registers. We take advantage of the fact that data points are sorted in the indexing phase and are laid out consecutively in the memory. Points located close to each other spatially are also placed together in the memory. Hence, while we compute the neighbours of each point in a subspace, most of the neighbours are likely to be loaded on the cache memory already as the potential neighbours of points within a subspace allotted to the processor are likely to be the same. Our measurements of the cache hit rate using the Performance Application Programming Interface (PAPI) tool [10] while analyzing the Bremen point cloud dataset [6] revealed an average hit rate of 98% for the L1 cache and 77% hit rate for the L2 cache, implying that only 0.46% of the memory access was from the L3 cache or the RAM. A detailed explanation of the Bremen point cloud dataset is given in the section on experimental evaluation.

With the points and their coordinates placed consecutively in the memory, the dimensions of the different points could be efficiently loaded into the vector registers. Figure 3 illustrates the memory layout of a three-dimensional dataset and how the vector registers can be used to efficiently vectorize the Euclidean distance computation.

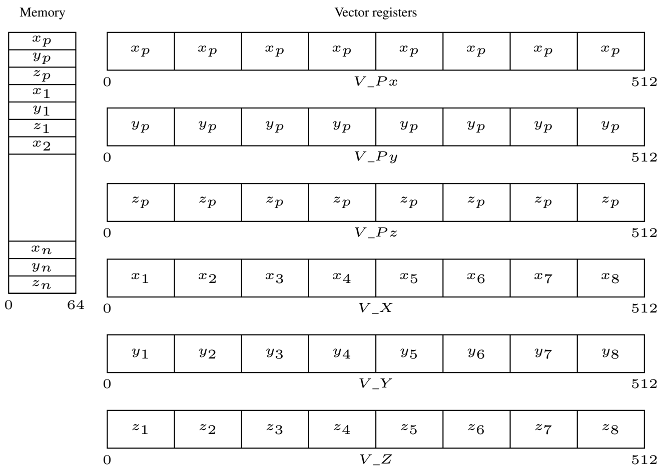

Memory
Vector registers
xp
yp
zp
$x1$
$y1$
$z1$
$x2$
xn
yn
zn
0
64
xp xp xp xp xp xp xp xp
V_Px
yp yp yp yp yp yp yp yp
V_Py
zp zp zp zp zp zp zp zp
V_Pz
$x1$ $x2$ $x3$ $x4$ $x5$ $x6$ $x7$ $x8$
V_X
$y1$ $y2$ $y3$ $y4$ $y5$ $y6$ $y7$ $y8$
V_Y
$z1$ $z2$ $z3$ $z4$ $z5$ $z6$ $z7$ $z8$
V_Z
0
512

FIGURE 3: Memory layout of a three-dimensional dataset

In Figure 3, registers V_X, V_Y and V_Z are vector registers that store the x, y and z values of the j data points in each iteration. Note that j depends on the number of elements that can fit in the vector register. The x, y and z coordinates of point p, from which the distances to its neighbours must be computed, are broadcasted into vector registers V_Px, V_Py and V_Pz respectively. To compute the Euclidean distance, we need to execute the following equation in the distance computation loop using SIMD instructions.

$$V_{dist} = (V\_X - V\_Px)^2 + (V\_Y - V\_Py)^2 + (V\_Z - V\_Pz)^2 \qquad (1)$$

Now that we have the distances stored in the V_{dist} register, we need to check if it is less than or equal to ϵ, and for those points that conform to the distance criteria we need to check if they are core points. Finally, the lowest cluster label of all the conforming core points is returned as the cluster's label. Cluster labels are signed 64-bit integers, and core points are identified by storing their cluster labels as negative. HPDBSCAN stores the points that lay within ϵ distance in a C++ `std::vector` $potential\_Nb$ which is dynamically expanded inside the for loop in Algorithm 3 using the `std::vector::push_back()` function. Reallocation of memory inside a tight loop will significantly hamper performance. Instead, VHPDBSCAN allocates memory for the vector before the loop begins by specifying the size of the vector to be n where n is the total number of potential neighbours and hence reserving the maximum memory possible. We fill the vector with an arbitrary constant INT_MAX and use a counter to keep track of the total number of points that are within ϵ distance. After the execution of the loop, only those indices of $potential\_Nb$ that do not contain INT_MAX will indicate core points. This approach will lead to a higher memory usage but will speed up the execution time as we can completely avoid reallocating memory in each iteration besides making the code SIMD friendly. The conditions to check if the Euclidean distance is less than or equal to ϵ and if a point is a core point can be easily vectorized using vector compare instructions.

The optimized and vectorized *regionQuery* algorithm is illustrated below.

Algorithm 5: regionQuery

```
Data: Point $p$, Dataset $data$, Indexes $potential\_Nb$,
      $\epsilon^2$, $minPoints$, Clusters $cluster\_labels$,
      Dataset dimension $Dim$, Elements per Vector $k$
Result: $label, neighbours\_p$
for $i = 0; i \leq length(potential\_Nb); i = i + k$ do
    $predicate_v$ = number_of_active_elements();
    $indices_v$ = loadArray($predicate_v$,
      $potential\_Nb[i]$);
    $scaled\_indices_v$ = $indices_v$ * $Dim$;
    $distance_v$ = loadOne($predicate_v$, 0.0);
    for $d = 0; d \leq Dim$ do
        $point_v$ = loadArray($predicate_v$,
          $data[p * Dim + d]$);
        $scaled\_indices_v = scaled\_indices_v + d$;
        $other\_point_v$ = gatherValues($predicate_v$,
          $scaled\_indices_v, data$);
        $square_v$ = $(other\_point_v - point_v)^2$;
        $distance_v = distance_v + square_v$;
    end
    $mask_v$ = vectorCompare($predicate_v$,
      $distance_v \leq \epsilon^2$);
    $nearby\_points_v$ = gatherValues($mask_v$,
      $indices_v, cluster\_labels$);
    $labels\_less\_than\_zero_v = nearby\_points_v \leq$
      0;
    $label$ = min($labels\_less\_than\_zero_v$);
    $neighbours_p.add(mask_v, indices_v)$;
end
```

In Algorithm 5, $gatherValues()$ is a vector operation that fetches values from the array using the indices provided by the first argument,which is again a vector register, and loads the fetched values in another vector register. The $loadOne(x)$ function simply fills the entire vector register with the value passed to it, and $loadArray()$, reads the consecutive values given by an array address and writes them onto a vector register. All variables that are vectors are denoted using a subscript v. The `min()` function returns the minimum value in the vector register passed to it. HPDBSCAN uses `std::minmax()` to retrieve the minimum label and assign it to all the points that belong to a neighbourhood. Since we know that the regionQuery() function already returns the smallest cluster label, we can do away with the use of std::minmax() function in VHPDBSCAN and directly update the rules with the new cluster label. In the subsequent sections, before we explain the implementation details of our vectorized algorithm using ARM's SVE intrinsics and x86 based AVX-512 intrinsics, we briefly introduce the features of SVE and the implementation of the same in the A64FX CPU and the implementation of AVX-512 in Intel Xeon 8368 (Icelake) CPU.

B. VECTORIZATION ON THE A64FX PROCESSOR

With an ever-increasing demand for vector processing capability by HPC workloads, ARM introduced a vector-length agnostic SIMD architecture model called the SVE [29].

The SVE architecture provides 32 new scalable vector registers whose width is implementation-dependent. These registers are extensions of the 32 and 128-bit advanced SIMD registers (V0-V31). There are also 16 scalable predicate registers and a set of control registers that give each privilege level the ability to virtualize the width of the vector registers. The A64FX is the world's first processor to implement SVE for supercomputers [29]. It can perform single-precision and half-precision floating-point calculations, as well as 8-bit and 16-bit calculations with 512-bit wide SIMD. It has 48 calculation cores and two or four assistant cores and is claimed to have a theoretical peak performance of 3.3792 teraflops [1] in double-precision floating point calculations. The execution unit of the A64FX micro-architecture consists of two fixed-point functional units, two functional units for address computation, and simple fixed-point arithmetic. It has two floating point units for SVE instructions and a separate predicate unit for executing predicate arithmetic. Both the floating point units support a SIMD length of 512 bits and can execute a fused multiply and add (FMA) operation every cycle. Hence, each core can operate on 32 double-precision floating points or 64 single-precision elements per cycle. Also, the floating point units support half-precision and bfloat16 formats. The L1 cache processes load/store instructions and has a 64kib instruction cache and a 64kib data cache that is capable of executing two simultaneous 64 byte load instructions or a single 64 byte store instruction. Before attempting to vectorize the function, we checked if the compiler can vectorize at least the distance computation loop inside the *regionQuery* function. The pseudo-code to compute the Euclidean distance computation between *point* and *other_point* of dimension d is displayed in Algorithm 4.

We checked with GCC 12 and ARM CLANG compilers on the A64FX partition of the IRENE High-Performance Cluster. The source code was compiled using the `-mpcu=a64fx` flag with the `-O3` optimization flag. On studying the generated assembly using the `objdump` tool, we found that both the compilers failed to vectorize the loop using SIMD instructions. With the help of SVE intrinsics, the straightforward way to vectorize the loop would use predicates to check if the loop count is less than the number of dimensions and load the SVE register with the maximum number of elements that is possible for an iteration and compute the distance. The C++ code of the squared Euclidean distance between two points of single precision floating point whose coordinates are stored in C++ `std::vector` is given below.

```
]
/*a and b are of std::vector<float> datatypes
    */
for (uint32_t i = 0; i < dimensions; i +=
    svcntw())
{
    svbool_t pg = svwhilelt_b32(i, n);
    svfloat32_t va = svld1(pg, &a[i]);
    svfloat32_t vb = svld1(pg, &b[i]);
    svfloat32_t diff = svsub_f32_z(pg, va, vb);
    svfloat32_t square = svmul_f32_z(
        pg, diff, diff
    );
    sum += svaddv_f32(pg, square);
}
```

Listing 1: Euclidean distance computation vectorization using SVE intrinsics.

C. VECTORIZATION ON AVX-512

AVX stands for Advanced Vector Extensions. AVX-512 is an expansion of the AVX and AVX-2 instruction sets, designed by Intel, that can be found in both x86 and ARM architectures. It expands the previous register length of 256 to 512 bits. We compiled the vectorized code using AVX-512 intrinsics using the Intel ICC Compiler 2021.5.0.

Once we confirmed that the compiler was using the right instruction set for both architectures, we proceeded to vectorize HPDBSCAN using the algorithm described in the previous section. The complete C++ implementation of the VHPDBSCAN function using SVE and AVX-512 intrinsics for a single precision multidimensional floating point dataset can be found in our open GitHub repository[1].

D. OPTIMIZATION FOR THE CLUSTERING OF HIGH DIMENSIONAL DATASETS

We noticed that in the case of clustering of high dimensional datasets such as datasets with dimensions greater than 15, a significant percentage of time is spent in the distribution of points among the different MPI ranks. Table 1 shows the percentage of time spent in various stages of HPDBSCAN for the different datasets used in our evaluation, along with their dimensions. We give detailed explanations of the used datasets in section V-B. Local DBSCAN clearly is the most compute intense stage in the analysis of all datasets, accounting for nearly almost all the runtime, in the analysis of the Bremen point cloud and the Household power consumption datasets. However, in the analysis of the Bridge image dataset, around 33% of the runtime is spent in the computation of scores. This was expected as the number of dimensions in the Bridge image dataset being the highest at 16 among all the datasets used in our analysis and the subsequent exponential time and space complexity of the computation of scores based on the number of dimensions as mentioned in section II-B. Parallelizing the computation of scores for all the cells in order to make use of the available CPU cores will have a significant impact on the performance. We use shared memory parallelism, using OpenMP so that each thread computes the score for the cells assigned to it. While HPDBSCAN directly iterated through a hash map that consisted of pairs of cell number and the number of points inside the cell, to compute the neighbours for each cell, VHPDBSCAN copies the key values to a std::vector in order to parallelize the iteration. In order to compute the neighbouring cells, only the cell number is required. Parallelization of iteration through a vector using OpenMP is straightforward and faster. The overhead involved in copying of the key values from the hash map to a std::vector is insignificant compared to the gains in parallelizing the computation of scores.

V. EXPERIMENTAL EVALUATION

In this section, we will describe the methodology and findings of the experiments conducted to evaluate VHPDBSCAN. The main focus of the investigation is to show the performance improvements over HPDBSCAN. The performance evaluation of the optimizations on various datasets with respect to computation time, energy [12], and the parallel programming metric speedup were evaluated in detail. Speedup is defined as the ratio between the time taken for running software on one processor t_1 and the time taken for running software on n processors t_n.

$$speedup = t_1/t_n \tag{2}$$

[1]https://github.com/JosephArnold/vhpdbscan/tree/HPDBSCAN-OPTIMIZATION

Dataset	**Bremen**		**Household**		**Bridge**	
Dimensions	3		7		16	
	Time [%]	**Time [s]**	**Time [%]**	**Time [s]**	**Time [%]**	**Time [s]**
Computing dimensions	1.00e-2	1.700	1.00e-3	1.700e-2	1.360e-5	5.700e-4
Computing cells	1.300e-2	2.270e1	1.8e-2	3.180	7.700e-5	3.200e-3
Sorting points	1.850e-1	3.240e1	3.100e-2	5.480	5.450e-5	2.300e-3
Computing scores	1.000e-1	1.750e1	5.600e-1	9.900e1	3.314e1	1.398e3
Distributing points	5.300e-2	9.280e1	2.900e-4	5.100e-2	1.350e-6	5.600e-5
Local DBSCAN	9.860e1	1.727e4	9.935e1	1.756e4	6.680e1	2.818e3
Merging halos	1.480e-5	2.590e-3	3.290e-6	5.810e-4	3.660e-6	1.540e-4
Appyling rules	3.330e-4	5.810e-2	1.400e-3	2.000e-1	3.270e-6	1.380e-4
Recovering order	2.800e-1	5.044e1	1.800e-2	3.180	4.330e-5	1.820e-3

TABLE 1: Percentage of time spent in each of the stages of HPDBSCAN and the absolute time

We evaluated the performance of the vectorized code on a A64FX cluster of the Irene supercomputer provided by TGCC (Très Grand Centre de Calcul du CEA), a high-performance computing infrastructure aiming at hosting state-of-the-art supercomputers in France. The supercomputer offers three kinds of nodes: regular computing, large memory, and GPU. The ARM A64FX cluster used for regular computation consists of a total 80 nodes with 48 cores each. Each CPU operates at a frequency of 1.8 GHz and is equipped with 32 GB HBM2 memory with a bandwidth of 1TB per second. The 48 cores of each processor are provided by four Non Uniform Memory Access (NUMA) nodes with each NUMA node having 12 compute cores. Based on Fujitsu PRIMEHPC FX700 technology, the 80 single-socket DDR-less compute nodes are connected via Mellanox InfiniBand and are integrated into GENCI's Joliot-Curie supercomputer. The hardware was allocated to us for the EUPEX (European Exascale Computing) project as a software development vehicle (SDV) to optimize scientific and machine learning applications to exploit exascale machines.

A. SOFTWARE SETUP

The operating system running on IRENE is Red Hat Enterprise Linux version 8.8. All applications in the test have been compiled with GCC 12.2.0 using the optimization level `Ofast` and architecture flag `mcpu=a64fx`. The MPI distribution on IRENE is Open MPI version 4.0.5. For the compilation of HPDBSCAN and VHPDBSCAN, an HDF5 development library including headers and C++ bindings is required. For HDF5, We used the preinstalled version 1.12.0.

B. DATASETS

For the evaluation of the VHPDBSCAN, we have chosen real-world datasets of dimensions three, seven and sixteen to observe how the vectorization is affected on by the number of dimensions. We briefly describe the datasets used for our evaluation below.

1) Point cloud of Bremen's old town

This data was collected and made available by Dorit Borrmann and Andreas Nüchter from the Institute of Computer Science at the Jacobs University, Bremen, Germany [6]. It is a 3D-point cloud of the old town of Bremen. A point cloud is a set of points and its representative coordinate system that often models the surface of objects. The dataset was used to benchmark the original implementation of HPDBSCAN and was obtained in HDF5 format from B2SHARE. DBSCAN can be applied here to clean the dataset from noise or outliers and find distinct objects, represented as clusters, in the point cloud like houses, roads, or people. The whole point cloud contains roughly 81 million data points.

2) Household dataset

A dataset containing 4 years of metering of electric power consumption of a household using a one-minute sampling rate: minute-averaged kilowatt, volt, and ampere measurements for the household as a whole and sub-metering (e.g., electric water heater and air-conditioner) in watt-hours. The dataset has been donated by Hebrail and Berard from Electricité de France R&D to the UCI Machine Learning Repository [20]. Originally, this dataset was 9-dimensional, but after removing time and date timestamps, we obtained a 7-dimensional dataset. The zipped text file contains 2,075,259 lines of semicolon-separated values, but some sample lines contain missing values: we removed all lines containing a "?" as a value, resulting in 2,049,280 data points. The dimensions are in different units, hence we normalized them using min max normalization using the pandas[2] library in Python.

3) Image dataset

In order to evaluate the behavior of VHPDBSCAN on a higher dimensional dataset, we choose a image dataset which is an image of a bridge where the size of the image is 256*256 divided into 4*4 pixel blocks [16]. We refer to the dataset as *Bridge* in the plots. Hence each of the 4096 pixel blocks is represented by 16 dimensions. Like mentioned earlier, the exponential time and space complexity involved in the computation of neighbourhood cells, we limit the number of dimensions to 16. For example, evaluating with a dataset of dimension of 32 with single precision floats, the number of operations will be a staggering 1.8e15 consuming a memory space of 7PB.

[2]https://pandas.pydata.org/

C. SPEEDUP EVALUATION OF VHPDBSCAN

We benchmark the VHPDBSCAN application's speedup using all the three datasets mentioned above on the A64FX. Our principal methodological approach is thereby as follows. Each benchmark is run five times, measuring the application's wall time at the beginning and end of the `main()` function of the process with the MPI rank 0 and the OpenMP thread number 0. We first study the scalability of the applications's shared memory parallelization using OpenMP on a single node by increasing the number of cores from 1 up to 48 as a single A64FX node contains 48 cores. We increase the number of cores after each set of five runs with the base measurement on exactly one core of the node. For each set of five runs benchmark, we report the mean runtime. The speedup coefficient is calculated compared to the single-core run based on the mean values of the measurements for each processor count configuration. In order to ensure the maximum utilization of each OpenMP thread, we increased the OpenMP dynamic chunk size further from the original value of 40 up to 2048 where we saw the highest performance improvements. Also, a low chunk value would result in higher atomic *min* operations on the same memory location. Beyond 2048, we saw that the performance improvements due to shared memory parallelism began to stagnate. Also, further increase could result in an uneven division of workload. We also assessed the performance by only increasing the chunk size and keeping the original implementation intact on the A64FX CPU. The runtime only decreased by roughly 20% proving that floating point pipelines were not utilized to the maximum despite more work to the OpenMP threads. Having fixed the OpenMP chunk size, we now explain each of our scaling experiments in detail.

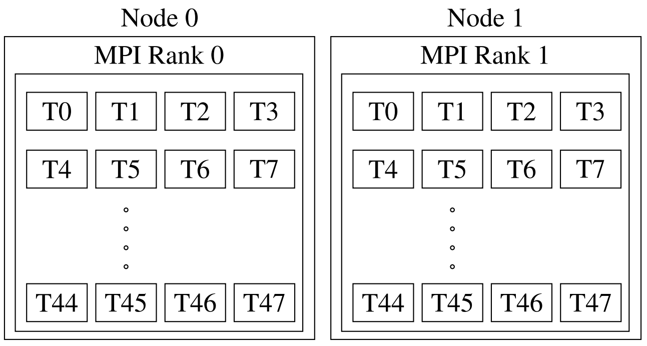


FIGURE 4: Hybrid parallelization using both MPI and OpenMP on two A64FX nodes each with 48 cores. Threads T0-T47 are each pinned to a core. Each MPI rank runs on a seperate A64FX node.

We first study the scalability of VHPDBSCAN on a single A64FX node by keeping the number of MPI ranks as one and increasing the number of OpenMP threads where each thread is assigned to a core. Figure 5 shows the runtimes of VHPDBSCAN and HPDBSCAN against the the number of cores. We can see that there is a reduction in the runtime by about 52% and 63% on the Bremen point cloud and Household consumption datasets respectively. In the case of the Bridge image dataset, improvements due to the parallelization of the local DBSCAN HPDBSCAN are seen only upto 8 cores after which the due to lack of parallelization of the computation of scores which is the main computational bottleneck results in no further drop in the runtime with the increase in the number of cores. However a clear reduction in the runtime is observed in the case of VHPDBSCAN thanks to the parallelization of the computation of scores. Also another important observation to make is that there is no significant improvement due to vectorization as the runtime of VHPDBSCAN is nearly the same as HPDBSCAN for a single core run in the case of the image dataset. Interestingly, the performance of the naive approach to the vectorization of the distance computation as mentioned in Algorithm 4 was similar to that of our approach. Compared to the Bremen and the Household power consumption datasets where the dimensions were three and seven respectively, the number of unused lanes in the SVE register in the case of the naive approach is much lesser (16 in the case of single precision floats). Ideally if the number of dimensions of the dataset was a multiple of the vector length, the naive approach may be even better than our approach as there will be no unused lanes in the SVE . Also, in our approach there is an additional overhead due to the broadcasting of the dimensions of point p whose neighbours are being computed $(n/sizeof(vectorregister)) * dimensions$ times where n is the number of points in the neighbouring cells of point p and the vector gather instruction, that must be executed for the same number of times to load the coordinates of the point from different memory locations. Note that the ARM microarchitecture manual[3] mentions that the number μ ops needed for a gather instruction is 11 whereas in the case of a SVE load instruction, it is only 5. However, this could not be experimentally verified due to the exponential time complexity involved in the computation of scores for higher dimensional datasets as mentioned in Section II-B. Figure 8 illustrates the scaling behavior on a single node. We observe that in the case of the Bremen point cloud dataset, both HPDBSCAN and VHPDBSCAN scale well up to about 8 cores, after which there is a gradual drop in the scalability. The speedup of the VHPDBSCAN version remains slightly higher than that of HPDBSCAN till up to 32 cores indicating higher utilization of the processors' vector registers. Also, for efficient usage of the SIMD capability of the processor, the number of points that can be loaded into the vector registers must be sufficiently large to offset the vectorization overhead. A key scalability limiting factor is that with an increase in the number of threads, there is a growing number of `atomicMin()` clashes. On the Household power consumption dataset, the scalability of both the VHPDBSCAN and the HPDBSCAN is much better. Note that the distance computations involved is highly dependent on the input pa-

[3]https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_Microarchitecture_Manual_en_1.3.pdf

rameters and the spatial distribution of a dataset which can subsequently influence the scaling behavior as well. Also, the Household dataset has higher dimensions than that of the Bremen point cloud dataset and would require more vector registers and SIMD instructions. With the increase in the number of cores, there is a better usage of floating point pipelines of all the cores involved. In the case of the Bridge image dataset, the scalability of VHPDBSCAN almost close to the ideal scalability up to 18 cores due to the parallelization of the computation of scores. Note that the computation of local DBSCAN is no longer the computational bottleneck and the increase in the number of cores has no impact on the performance which is clearly reflected in the poor scalability on the analysis of image dataset in HPDBSCAN.

In order to study the scalability of VHPDBSCAN and HPDBSCAN using MPI parallelization alone, we use one MPI process per core while keeping the number of OpenMP threads to one. Figure 6 and Figure 9 illustrate the runtime and the scaling behavior of VHPDBSCAN and HPDBSCAN using only MPI parallelization respectively. Figure 6 shows a 52% reduction in the execution time in the analysis of both the Bremen point cloud and the Household power consumption datasets. However, in both the datasets, VHPDBSCAN shows lower scalability than HPDBSCAN. With the increase in the number of MPI ranks, the points assigned for clustering to a single core decreases. With a large number of MPI ranks assigned to VHPDBSCAN, in the MPI only parallelization, the number of points assigned to a core decreases as a result of which the performance benefits due to vectorization is offset by the MPI synchronization overhead and also the vectorization overhead. However, in the case of HPDBSCAN, the time spent in the local DBSCAN computation continues to be more significant than the MPI synchronization overhead resulting in better scalability. This observation is in line with Amdahl's law. In the benchmark, we use a constant problem size, disallowing infinite speedup performance gains. In the case of the Bridge image dataset, similar to our observation in the previous experiment, there is no significant improvement in the performance due to the vectorization of the local DBSCAN. Also, due to the lack of OpenMP parallelization of the computation of scores, the runtime should have been similar to of HPDBSCAN where the computation of scores is not parallelized anyways. However, we do observe a noticeable reduction in the runtime in VHPDBSCAN which can be attributed to the reduction in the runtime in the merging of the halo cells. MPI's `MPI_alltoallv` API is used to exchange the indices of the halo cells among all the MPI processes. In the case of VHPDBSCAN, the indices are stored as 32-bit signed integers unlike in HPDBSCAN where they are stored as 64-bit signed integers. This results in lower memory bandwidth in MPI communication. Note that the SVE intrinsics require single precision floats to be accessed using 32-bit integers and double precision floats accessed using 64-bit indices. As the image dataset was represented using 32-bit floats, the indices also had to be 32-bit integers accordingly.

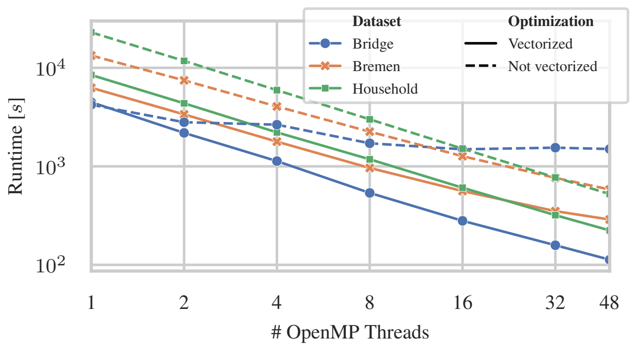


FIGURE 5: Runtime against the number of cores on a single A64FX node analyzing the Bremen, Household and Bridge pixel datasets.

Finally, we study the scaling behavior of VHPDBSCAN and HPDBSCAN by enabling both MPI parallelization and OpenMP parallelization. Figure 4 illustrates the set up for hybrid parallelization on two A64FX nodes, each running a MPI rank. We achieve this by assigning each MPI rank to a A64FX node and parallelizing the computation locally on each node by assigning each OpenMP thread to a core. The number of OpenMP threads per MPI rank is 48 as each A64FX node has only 48 cores. Figure 7 shows a 51% and a 49% reduction in the runtimes for the Household power consumption dataset and the Bremen point cloud dataset respectively, similar to what we observed in the case of OpenMP only and MPI only parallelization experiments. The scalability of VHPDBSCAN trends lower than that of HPDBSCAN on both the datasets with the gap widening further in the case of Bremen point cloud dataset as shown in Figure 10. With a higher number of workers being assigned to exploit both OpenMP and MPI parallelism and the problem size remaining constant, the effect of Amdahl's law becomes more pronounced. The deviation from the ideal scaling behaviour is the highest in the case of hybrid (both OpenMP and MPI parallelism. A similar observation can be seen using the Bridge image dataset and the only difference is that all the performance improvements using VHPDBSCAN come from the parallelization of the computation of scores. The lack of scalability in HPDBSCAN is simply due to the fact that there is no OpenMP parallelization in the computation of scores which is the most compute intensive function in the case of the image dataset. The lack of scalability in the case of VHPDBSCAN can be attributed to the effect of Amdahl's law as the problem size is very small compared to that of the Bremen point cloud and the Household power consumption datasets as our main objective in the analysis of the Bridge image dataset was to evaluate the behavior of our application on a dataset of a dimension that is almost equal to the number of vector registers. Note that the runtime in the case of VHPDBSCAN is around 10 times less than that of HPDBSCAN.

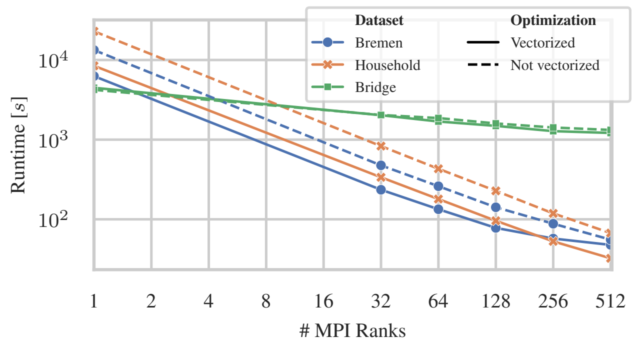


FIGURE 6: Runtime against the number of cores on a single A64FX node analyzing the Bremen, Household and Bridge pixel datasets.

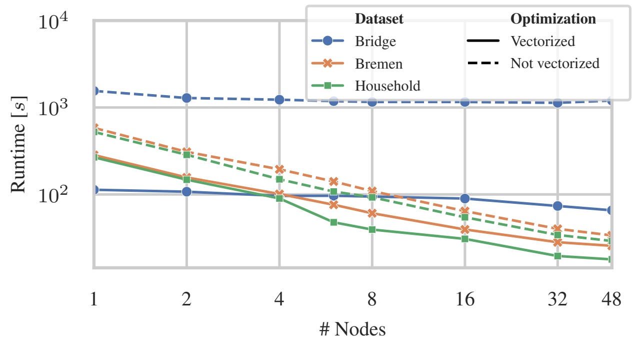


FIGURE 7: Runtime against the number of nodes, each node utilizing all the 48 cores in the analysis of the Bremen and Household datasets.

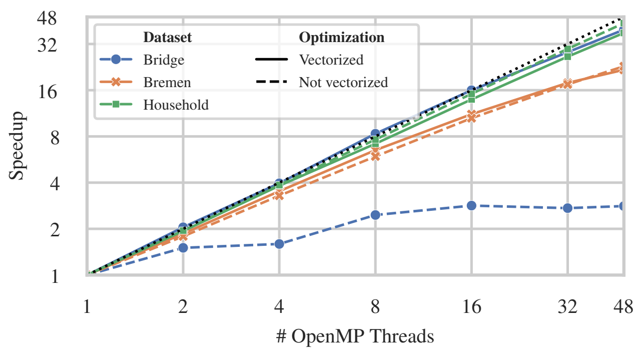


FIGURE 8: Strong scaling number of cores on a single A64FX node analyzing the Bremen and Household datasets.

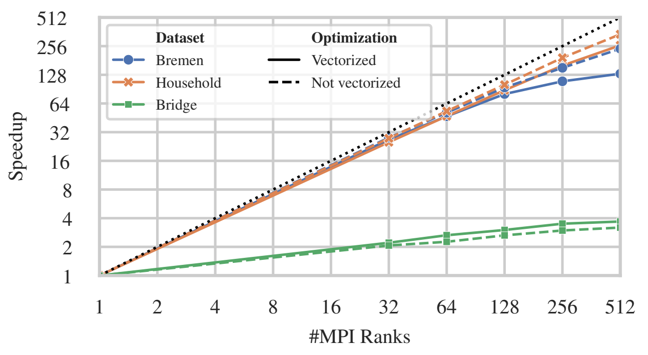


FIGURE 9: Strong scaling using MPI only parallelization analyzing the Bremen and Household datasets.

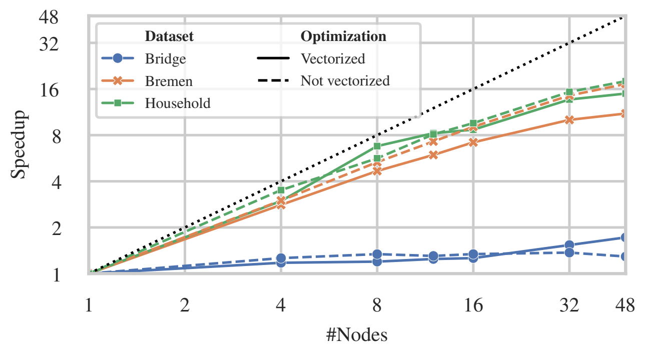


FIGURE 10: Strong scaling using both OpenMP and MPI parallelization analyzing the Bremen and Household datasets.

D. ENERGY CONSUMPTION

Reporting and improving the energy efficiency of hardware and software has seen increased interest in recent years in the HPC community. The interest grew even more since large language models became popular, after many publications reported the estimated carbon emissions of training and using such models in HPC systems and cloud data centers [30]. Since then, efforts to quantify the energy cost of HPC applications have increased, leading to the development of energy monitoring tools targeted at the HPC environment [17]. We report the CPU energy consumption of HPDBSCAN and VHPDBSCAN to observe how the energy scaling behavior changes with the number of parallel threads on a processor. We report energy and not CO_2, as we believe energy provides a more comparable value across hardware and applications, and does not have any temporal and geographical dependencies like CO_2 [14].

We measured the energy consumption of two CPUs while running HPDBSCAN and VHPDBSCAN: first on the A64FX as mentioned in the previous experiments, and secondly on the Intel Xeon Platinum 8368 [21]. We made use of the HoreKa Blue super-computer at the Super Computing Centre in Karlsruhe. Each of the 610 nodes in HoreKa Blue contains two Intel Xeon Platinum 8368 processors, a 960 GB SSD, and is interconnected with an InfiniBand HDR fabric. Out of the 610 nodes, 32 nodes are high memory nodes with 512 GB of main memory, and 8 nodes are extra large memory nodes 4096 GB.Intel Xeon Platinum includes the AVX512 extension, a set of ultrawide 512 registers capable of vectorizing operations of a variety of data formats. AVX512 has been shown to improve performance in different types of applications [8], [11]. We translated the SVE implementation to AVX512 to compare the performance and the energy consumption between VHPDBSCAN and HPDBSCAN algorithm. The total energy consumption of the CPU was collected using the `perf` tool[4]. `perf` is a Linux tool that allows users to collect per process and system-wide performance counters while an application is running. On the A64FX, we used the EA_CORE, EA_L2, and EA_MEMORY performance monitoring unit (PMU) events to measure energy, as reported by Fujitsu[5]. These registers track the energy used by the individual cores, the L2 cache shared by a Core Memory Group (CMG), and the High Bandwidth Memory of each CMG in the CPU. Since the `perf` tool collects the PMU events for each core whereas EA_L2 and EA_MEMORY account for a CMG consisting of 12 cores, we only monitored the EA_L2 and EA_MEMORY values for each core in a CMG. The energy data on the Intel Xeon Platinum 8368 was collected using the `power/energy-pkg/` event, which reports a mostly complete picture of "Core" and "Uncore" components of the CPU, based on the Intel RAPL interface [23].

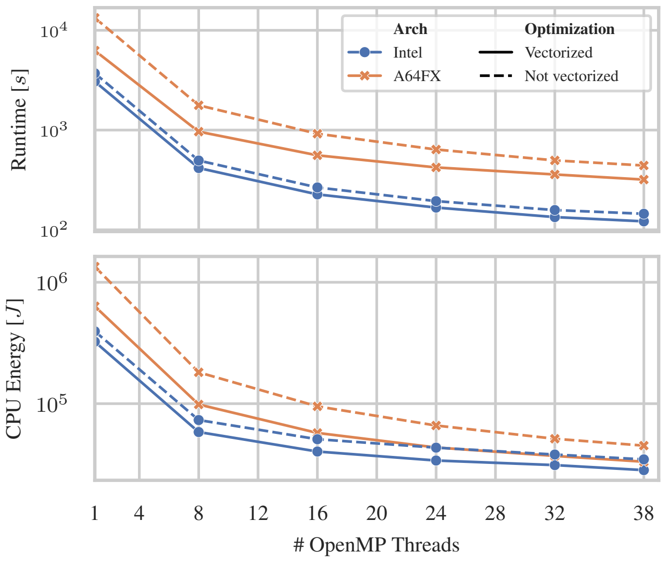


FIGURE 11: Runtime and energy comparison between the vectorized and non vectorized version of HPDBSCAN running on the A64FX and the Intel Xeon processors.

An initial observation from Figure 11 reveals that energy follows the same trend as the runtime, with a decrease in the total consumed energy by the CPUs as the clustering runtime decreases. Secondly, it can be seen that, while there is a clear performance improvement between VHPDBSCAN and HPDBSCAN running on the Intel processor, it has a lesser impact than can be seen on the A64FX. By looking at the assembly code of the HPDBSCAN with the Intel compiler, we found that it was making use of vectorized registers of lower length (AVX, AVX2), so the potential gains by using the longer vector registers are lower.

Thirdly, one interesting difference between the two architectures is the changes in power draw as the number of OpenMP threads increases. On the Intel Xeon processors, the power draw goes from 105 Watts when running HPDBSCAN on a single core, to 235 Watts when using all 38 cores, not too far from reaching the advertised Thermal Design Power of 270 Watts[6]. On the other hand, based on the energy data gathered from the A64FX, the total power draw goes from 100 Watts to 104 Watts ($\approx$ 86 Watts due to the compute cores, and the rest due to the shared L2 Cache and HBM), with no major increase due to the higher core utilization.

VI. CONCLUSION

In this paper, we proposed a new algorithm for the vectorization of the DBSCAN algorithm by taking advantage of the memory layout of the points to fill the SIMD registers. We also showed how parallelization of the computation of scores for each cell in the grid is critical for speedup in the analysis of higher dimensional datasets. While the existing HPDBSCAN implementation was already scalable at both the shared memory and the distributed memory levels, we demonstrated that our proposed VHPDBSCAN could further exploit the vector instructions of modern processors to gain significant performance improvements. We showed performance improvements due to vectorization on two different datasets on the A64FX processor and demonstrated that VHPDBSCAN outperformed the HPDBSCAN code by a factor of two. Apart from performance improvements, we also analysed the energy gains due to our optimizations on both architectures. We investigated the energy consumption of VHPDBSCAN compared to HPDBSCAN, where we observed a reduction in the total energy used by the processor proportional to the reduction in runtime, 50% on the A64FX processor, and 19% on the Intel Xeon processor. In the future, we plan to extend VHPDBSCAN to support other data types supported by modern CPUs, such as half-precision floating point and float data types. This would result in lower memory bandwidth for applications that do not demand the accuracy of current 32 or 64-bit floating point formats. Additionally, we plan to offload the computationally intensive task of distance calculation to various hardware accelerators, including GPUs.

[4]https://perf.wiki.kernel.org/

[5]https://github.com/fujitsu/A64FX/blob/master/doc/A64FX_PMU_Events_v1.3.pdf

[6]https://ark.intel.com/content/www/us/en/ark/products/212455/intel-xeon-platinum-8368-processor-57m-cache-2-40-ghz.html

REFERENCES

[1] FUJITSU Processor A64FX — fujitsu.com. https://www.fujitsu.com/global/products/computing/servers/supercomputer/a64fx/#anc-03. [Accessed 07-03-2024].

[2] Parallel implementation of statistical dbscan algorithm for spark-based clustering on google cloud platform. International Journal of Intelligent Engineering and Systems, 2023.

[3] Natvig Lasse Al Hasib Abdullah, Cebrian Juan M. A vectorized k-means algorithm for compressed datasets: design and experimental analysis. The Journal of Supercomputing, 74:2705– 2728, 2018.

[4] Hadian Ali and Saeed Shahrivari. High performance parallel k-means clustering for disk-resident datasets on multi-core cpus. The Journal of Supercomputing, 69(2):845–863, 2014.

[5] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. ACM Comput. Surv., 11(4):397–409, dec 1979.

[6] C. Bodenstein. HPDBSCAN benchmark test files.

[7] Surekha Borra, Rohit Thanki, and Nilanjan Dey. Satellite Image Clustering, pages 31–52. Springer Singapore, Singapore, 2019.

[8] Berenger Bramas. A novel hybrid quicksort algorithm vectorized using AVX-512 on intel skylake. International Journal of Advanced Computer Science and Applications, 8(10), 2017.

[9] Stefan Brecheisen, Hans-Peter Kriegel, and Martin Pfeifle. Parallel density-based clustering of complex objects. In Wee-Keong Ng, Masaru Kitsuregawa, Jianzhong Li, and Kuiyu Chang, editors, Advances in Knowledge Discovery and Data Mining, pages 179–188, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[10] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl., 14(3):189–204, aug 2000.

[11] Juan M. Cebrian, Lasse Natvig, and Magnus Jahre. Scalability analysis of AVX-512 extensions. The Journal of Supercomputing, 76(3):2082–2097, 2020.

[12] Debus Charlotte, Piraud Marie, and Götz Markus Streit Achim, Theis Fabian. Reporting electricity consumption is essential for sustainable ai. Nature Machine Intelligence, 5(11):1176–1178, 2023.

[13] Massimo Coppola and Marco Vanneschi. High-performance data mining with skeleton-based structured parallel programming. Parallel Computing, 28(5):793–813, 2002.

[14] Charlotte Debus, Marie Piraud, Achim Streit, Fabian Theis, and Markus Götz. Reporting electricity consumption is essential for sustainable AI. Nature Machine Intelligence, 5(11):1176–1178, 2023. Number: 11 Publisher: Nature Publishing Group.

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, KDD'96, pages 226–231. AAAI Press, 1996.

[16] P. Fränti, M. Rezaei, and Q. Zhao. Centroid index: cluster level similarity measure. Pattern Recognition, 47(9):3034–3045, 2014.

[17] Juan Pedro Gutiérrez Hermosillo Muriedas, Katharina Flügel, Charlotte Debus, Holger Obermaier, Achim Streit, and Markus Götz. perun: Benchmarking energy consumption of high-performance computing applications. In José Cano, Marios D. Dikaiakos, George A. Papadopoulos, Miquel Pericàs, and Rizos Sakellariou, editors, Euro-Par 2023: Parallel Processing, Lecture Notes in Computer Science, pages 17–31. Springer Nature Switzerland, 2023.

[18] Markus Götz, Christian Bodenstein, and Morris Riedel. HPDBSCAN: highly parallel DBSCAN. In Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments, MLHPC '15, pages 1–10. Association for Computing Machinery, 2015.

[19] Jiawei Han, Micheline Kamber, and Jian Pei. 11 - advanced cluster analysis. In Jiawei Han, Micheline Kamber, and Jian Pei, editors, Data Mining (Third Edition), The Morgan Kaufmann Series in Data Management Systems, pages 497–541. Morgan Kaufmann, Boston, third edition edition, 2012.

[20] Georges Hebrail and Alice Berard. Individual household electric power consumption. Published: UCI Machine Learning Repository.

[21] Intel. Intel® xeon® platinum 8368 processor (57m cache, 2.40 GHz) product specifications.

[22] Eshref Januzaj, Hans-Peter Kriegel, and Martin Pfeifle. Scalable density-based distributed clustering, 2014.

[23] Kashif Khan, Mikael Hirki, Tapio Niemi, Jukka Nurminen, and Zhonghong Ou. RAPL in action: Experiences in using RAPL for power measurements. ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS), 3, 2018.

[24] Sonal Kumari, Poonam Goyal, Ankit Sood, Dhruv Kumar, Sundar Balasubramaniam, and Navneet Goyal. Exact, fast and scalable parallel dbscan for commodity platforms. In Proceedings of the 18th International Conference on Distributed Computing and Networking, ICDCN '17, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Md. Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. A new scalable parallel dbscan algorithm using the disjoint-set data structure. In SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pages 1–11, 2012.

[26] Timofey Rechkalov and Mikhail Zymbler. A study of euclidean distance matrix computation on intel many-core processors. In Leonid Sokolinsky and Mikhail Zymbler, editors, Parallel Computational Technologies, pages 200–215, Cham, 2018. Springer International Publishing.

[27] Aditya Sarma, Poonam Goyal, Sonal Kumari, Anand Wani, Jagat Sesh Challa, Saiyedul Islam, and Navneet Goyal. μdbscan: An exact scalable dbscan algorithm for big data exploiting spatial locality. In 2019 IEEE International Conference on Cluster Computing (CLUSTER), pages 1–11, 2019.

[28] Bao-Quan Shi, Jin Liang, and Qing Liu. Adaptive simplification of point cloud using k-means clustering. Computer-Aided Design, 43(8):910–922, 2011.

[29] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The arm scalable vector extension. IEEE Micro, 37(2):26–39, March 2017.

[30] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. Number: arXiv:1906.02243.

[31] Yiqiu Wang, Yan Gu, and Julian Shun. Theoretically-efficient and practical parallel dbscan. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, page 2555–2571, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Alan Woodley, Ling-Xiang Tang, Shlomo Geva, Richi Nayak, and Timothy Chappell. Parallel k-tree: A multicore, multinode solution to extreme clustering. Future Generation Computer Systems, 99:333–345, 2019.

[33] Roung-Shiunn Wu and Po-Hsuan Chou. Customer segmentation of multiple category data in e-commerce using a soft-clustering approach. Electronic Commerce Research and Applications, 10(3):331–341, 2011.

[34] Kriegel Hans-Peter Xu Xiaowei, Jäger Jochen. A fast parallel clustering algorithm for large spatial databases. Data Mining and Knowledge Discovery, 3:263–290, 1999.

JOSEPH ARNOLD XAVIER received his Bachelor degree in Computer Science and Engineering from Visvesvaraya Technological University (VTU) and Masters in Computer Science and Engineering from the National Institute of Technology Puducherry, India. His work focuses primarily on optimization of applications in high Performance computing systems. He is currently pursuing his Ph.D degree in Computational Engineering from the University of Iceland in conjunction with the Juelich Supercomputing Centre, Germany.

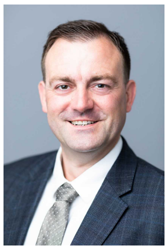
JUAN PEDRO GUTIÉRREZ HERMOSILLO MURIEDAS received his B.Sc. in Electrical Engineering and M.Sc. in Computer Science from the Karlsruhe Institute of Technology. At the present moment, he is pursuing a Ph.D. degree at the Scientific Computing Centre in Karlsruhe, Germany. His work focuses on data-distributed applications in high-performance computing systems, with special interest in finding efficient parallelization strategies, as well as monitoring and benchmarking the energy consumption of large-scale applications.

STEPAN NASSYR recieved his B.Sc. and M.Sc. in Physics at the University of Wuppertal (Bergische Universität Wuppertal) and is currently pursuing a Ph.D. degree in Computer Science from the University of Wuppertal in conjunction with the Jülich Supercomputing Centre in Germany. His work focuses on microarchitectural analysis and co-design with a focus on ARM and RISCV processors, employing cycle-level simulation and design space exploration.

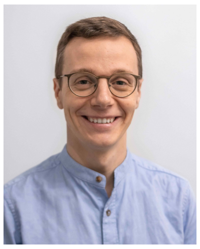
ROCCO SEDONA (Member, IEEE) received his B.Sc. and M.Sc. degrees in Information Engineering from the University of Trento in 2016 and 2019, respectively, and a Ph.D. degree in Computational Engineering from the University of Iceland in 2023. He is a member of the “AI and ML for Remote Sensing” Simulation and Data Lab at the JSC in Germany. His research interests primarily lie in the field of Deep Learning and its application to Remote Sensing data. He has extensively utilized optical satellite data acquired by Landsat (NASA) and Sentinel (ESA) missions towards near real-time land cover classification. Additionally, he specializes in Distributed Deep Learning on High-Performance Computing systems, an area of study that he has been actively engaged in since 2019.

MARKUS GÖTZ (Member, IEEE) received his B.Sc. and M.Sc. degrees in IT-System Engineering from the University of Potsdam, Germany, in 2010 and 2014 respectively, with intermediate stays the Blekinge Tekniska Högskola, Sweden and CERN, Switzerland. Since 2017 he holds a Ph.D. degree in Computational Engineering from the University of Iceland in conjunction with the Juelich Supercomputing Centre, Germany. Currently, he is a post-doctoral researcher at the Scientific Computing Centre, Karlsruhe Institute of Technology, Germany as the project manager for the Helmholtz Analytics Framework and the head of the Helmholtz AI consultants team. In line with his work, he focuses on applied artificial intelligence and data analysis on high-performance cluster systems to work on the grand challenges in the natural sciences. Markus Götz’s research interests include machine learning, global optimization as well as parallel algorithm engineering. He is a member of the IEEE.

ACHIM STREIT is one of the directors of the Steinbuch Centre for Computing at the Karlsruhe Institute of Technology (KIT). He is also a Professor for Distributed and Parallel High-performance Computing Systems at KIT’s department of Informatics. In 1999, he received a Diploma in Computer Science from the University of Dortmund, Germany and in 2003 a Ph.D. degree in the same subject from the University of Paderborn, Germany. Afterwards Achim Streit led the Federated Systems and Data Division at the Juelich Supercomputing Centre, Germany. He initiated and chaired several national and international research initiatives within the Helmholtz association (e.g. Helmholtz Data Federation and Helmholtz Information & Data Science Academy (HIDA)) on the national level (e.g. NFDI4Ing and NFDI-MatWerk) and the European level (e.g. EUDAT and EOSC). His research interests include high-performance and data-intensive computing, Big Data and federated data management, data analytics as well as job scheduling and resource management for parallel and distributed computing.



MORRIS RIEDEL (Member, IEEE) received his PhD from the Karlsruhe Institute of Technology (KIT) and worked in data-intensive parallel and distributed systems since 2004. He is currently a Full Professor of High-Performance Computing with an emphasis on Parallel and Scalable Machine Learning at the School of Natural Sciences and Engineering of the University of Iceland. Since 2004, Prof. Dr. - Ing. Morris Riedel held various positions at the Juelich Supercomputing Centre of Forschungszentrum Juelich in Germany. In addition, he is the Head of the joint High Productivity Data Processing research group between the Juelich Supercomputing Centre and the University of Iceland. Since 2020, he is also the EuroHPC Joint Undertaking governing board member for Iceland. His research interests include high-performance computing, remote sensing applications, medicine and health applications, pattern recognition, image processing, and data sciences, and he has authored extensively in those fields. Prof. Dr. – Ing. Morris Riedel online YouTube and university lectures include High-Performance Computing – Advanced Scientific Computing, Cloud Computing and Big Data – Parallel and Scalable Machine and Deep Learning, as well as Statistical Data Mining. In addition, he has performed numerous hands-on training events in parallel and scalable machine and deep learning techniques on cutting-edge HPC systems.



GABRIELE CAVALLARO (Senior Member, IEEE) received his B.Sc. and M.Sc. degrees in Telecommunications Engineering from the University of Trento, Italy, in 2011 and 2013, respectively, and a Ph.D. degree in Electrical and Computer Engineering from the University of Iceland, Iceland, in 2016. From 2016 to 2021 he has been the deputy head of the "High Productivity Data Processing" (HPDP) research group at the Jülich Supercomputing Centre (JSC), Forschungszentrum Jülich, Germany. Since 2022, he is the Head of the "AI and ML for Remote Sensing" Simulation and Data Lab at the JSC and an Adjunct Associate Professor with the School of Natural Sciences and Engineering, University of Iceland, Iceland. From 2020 to 2023, he held the position of Chair for the High-Performance and Disruptive Computing in Remote Sensing (HDCRS) Working Group under the IEEE GRSS Earth Science Informatics Technical Committee (ESI TC). In 2023, he took on the role of Co-chair for the ESI TC. Concurrently, he serves as Visiting Professor at the Φ-lab within the European Space Agency (ESA), where he contributes to the Quantum Computing for Earth Observation (QC4EO) initiative. Additionally, he has been serving as an Associate Editor for the IEEE Transactions on Image Processing (TIP) since October 2022. He was the recipient of the IEEE GRSS Third Prize in the Student Paper Competition of the IEEE International Geoscience and Remote Sensing Symposium (IGARSS) 2015 (Milan - Italy). His research interests cover remote sensing data processing with parallel machine learning algorithms that scale on distributed computing systems and innovative computing technologies.