High-Performance Visualization of Multi-Dimensional Gene Expression Data

Marjan Trutschl*, Phillip C.S.R. Kilgore†‡, and Urška Cvek†∥

* Department of Computer Science, LSU Shreveport
† Laboratory for Advanced Biomedical Informatics, LSU Shreveport
‡ Center for Molecular and Tumor Virology, LSU Health Sciences Center - Shreveport
Email: * mtrutsch@lsus.edu, † pkilgore@lsus.edu, ∥ ucvek@lsus.edu

Abstract—Previous application of Kohonen’s self organizing map to common visualizations has yielded promising results. In this research, we extend the classic two-dimensional scatter plot visualization algorithm into the third dimension by permitting competition to occur within a three-dimensional search space. This approach takes advantage of spatial memory and increases the intrinsic dimensionality of a widely used visualization technique. We also present a method of parallelizing this novel algorithm as a method of overcoming the runtime complexity associated with it using MPI. We note that this algorithm responds extremely well to parallelization and that it leads to an effective method for knowledge discovery in complex multidimensional datasets.

Index Terms—Machine Learning, Neural Networks, Information Visualization, Parallel Architectures, Clustering Algorithms

I. INTRODUCTION

The search for an effective visualization that results in insight of multidimensional data has been an ongoing endeavor for more than a century and a half. Two notable early examples include Charles Minard’s flow map, which dramatically depicts the defeat of Napoleon’s Russian campaign of 1812 [22], and Nightingale’s rose, used to show that preventable diseases contributed greatly to mortality in the Crimean War [23]. Later, other methods (such as Parallel Coordinates [13], RadViz [12], and SmartJitter [9], [28]) have been presented as viable options for visualizing multidimensional data.

While SmartJitter was initially implemented as a two-dimensional plot [28], a three-dimensional (3D) implementation of SmartJitter exhibits enticing properties since 3D visualizations have been shown to take advantage of spatial memory [6]. Previous work has shown a 3D implementation of SmartJitter which treats the map as a volume instead of a grid [9]. However, in some cases users want to focus on two well-understood dimensions in a multi-dimensional dataset while preserving the remaining n-2 dimensional values.

In this paper, we propose an intermediate approach. We start with a 2-dimensional (2D) SmartJitter resembling a scatter plot and then extend the plot into an artificially generated third dimension consisting of self-organized stacks of data. This results in a visualization offering the ease of interpretation that comes with a 2D scatter plot and at the same time provides additional information in the third dimension, reflecting the multi-dimensional data correlation along m selected dimensional values where 0 ≤ m ≤ n - 2.

We begin with a brief background on Kohonen’s self-organizing map and the SmartJitter algorithm that is based upon it, followed by the implementation of the extrusion of SmartJitter into the third dimension in Section III. Section IV evaluates the performance of the algorithm and offers a method of parallelization. After the analysis of experimental data in Section V, we discuss the results achieved and summarize the contribution of this work in Section VI.

II. BACKGROUND

The SmartJitter algorithm utilizes Kohonen’s self-organizing map [18] (SOM) to rearrange the input records of a scatter plot in such a way that records located close to each other are closely correlated [9], [17], [28]. Moreover, the algorithm coarsely preserves the topology of a scatter plot while permitting closely associated records to aggregate into spatial clusters based on chosen dimensional values in a multi-dimensional dataset [28].

Kohonen’s SOM is an unsupervised analogue of the artificial neural network. Though Kohonen postulated that self-organization need not require a neural mechanism, he found a neural model to be compelling, since it was known at the time that several mapping processes occurred in the brain [18], [20]. As such, Kohonen’s SOM may be described as a competitive learning process operating on a 2D grid of neurons [18].

Like artificial neural networks, SOMs are composed of neurons with an associated weight. In SOMs, each neuron is associated with a weight vector, which is a vector of the same length and within the range of the input set [18], and these neurons are sometimes referred to as output nodes [9]. There are two primary phases in a SOM: findWinner and updateNeighborhood. The findWinner routine finds an output node $\Omega_{\hat{i}}$ for input record $\hat{i}$ such that the distance between $\hat{i}$’s weight vector and $\hat{i}$ is minimal, as expressed in (1):

$$\exists \Omega_{\hat{i}} \in \Omega, \forall \Omega_{i} \in \Omega : ||\hat{i} - \Omega_{\hat{i}}|| \leq ||\hat{i} - \Omega_{i}||$$

(1)

Note that by (1), multiple candidate winning nodes may exist, yet exactly one winning node must be chosen, since records map to a single output node. Thus, we must have
a “tie-breaker” condition to alleviate this issue. We have specifically chosen a lexicographical ordering on the coordinate triple \([z, y, x]\) to serve as this tie-breaker, and a candidate winner whose coordinates are lexicographically less than all other candidate winners is chosen. A straightforward implementation of this routine may implicitly do this to begin with; however, this can become an issue when parallelizing the algorithm.

The result of (1) depends on the current state of the SOM. Our initialization method randomly initializes each weight vector’s dimensional value to fall in range for the corresponding dimensional value in the input dataset. Regardless of what method is used, however, no learning can take place until the SOM is updated to account for the error between the input record and the winning node; otherwise, \texttt{findWinner} will continually associate input records with the output nodes from the previous step. In order to rectify this issue, another routine called \texttt{updateNeighborhood} is performed after the winning node for an input record is determined.

The \texttt{updateNeighborhood} performs propagation of the error between the input record and winning node to the surrounding output nodes [9], [18]. This is achieved by selectively updating output nodes within the neighborhood of the winning node. During this phase, a neighborhood weight \( \omega \) is calculated from the current learning rate \( \lambda(t) \) and the current neighborhood radius \( r \). We calculate the neighborhood rate as a Gaussian function of time [9], [17], [28], as seen in (2):

\[
\omega(t, x, y, z) = \lambda(t) e^{-\frac{x^2+y^2+z^2}{\sigma^2}} \tag{2}
\]

The learning rate is a function of time, and is derived from \( \lambda_i \) and \( \lambda_f \), which are initial and final learning rates respectively. It is typically the case that \( \lambda_i >> \lambda_f \), since coarse adjustments are made early on. Instead of running until entropy is minimized, the algorithm executes within a specified number of epochs \( t \). The learning rate at time \( t \) is rendered as in (3) [9], [28]:

\[
\lambda(t) = \lambda_i \cdot \left( \frac{\lambda_f}{\lambda_i} \right)^{t/t_m} \tag{3}
\]

With a clear definition for both the neighborhood weight and learning rate, it is possible to account for the error previously discussed. The new value of a component \( c \) at time \( t + 1 \) for weight vector \( \Omega_k \) can be rendered as in (4) [9], [28]:

\[
\Omega_{k(c,t+1)} = \Omega_{k(c,t)} + \omega(t) \cdot (\vec{i}_k - \Omega_{k(c,t)}) \tag{4}
\]

Although \texttt{SmartJitter} is derived from Kohonen’s concept of the self-organizing map, it is different from Kohonen’s SOM in that competition is restricted to a subset of the map [9], [28]. In the case of \texttt{SmartJitter}, the map is partitioned at two levels. A primary partitioning scheme is used to approximate a scatter plot through a process called premapping (Fig. 1). In premapping, each input record is assigned by its component dimensions to a single cell within a grid of \( W_p \times H_p \) bins (5):

\[
[x', y'](i) = \left[ \left( W_p \cdot \frac{i_x - x_{min}}{x_{max} - x_{min}} \right), \left( H_p \cdot \frac{i_y - y_{min}}{y_{max} - y_{min}} \right) \right] \tag{5}
\]

In (5), \( i \) denotes an input record, while \( x \) and \( y \) denote component dimensions. The result is a zero-based point, and \([0, 0]\) denotes the bottom-left hand corner of the primary grid. The purpose of the primary grid is to preserve the topology of the scatter plot and provide a good approximation of the dataset. Kohonen permits all nodes to be candidates for selection during competition [18] (the \texttt{findWinner} phase), but \texttt{SmartJitter} only allows output nodes in the same primary grid cell to be considered [9], [28].

Within each primary grid cell, there is a \( W_s \times H_s \) secondary grid of output nodes. During the \texttt{findWinner} routine, the search space is composed of the output nodes contained within this region (Fig. 2). However, during the \texttt{updateNeighborhood} routine, the entire SOM is taken into consideration with neighborhood size decreasing as the learning takes place [9], [28]. A Kohonen SOM of dimensions \( W_k \times H_k \) may be expressed as a \texttt{SmartJitter} SOM with \([W_p, H_p] = [1, 1]\) and \([W_s, H_s] = [W_k, H_k]\), since using such topology would result in each record being premapped to the same primary grid cell.

Fig. 3 depicts a simplified and generic training algorithm used by \texttt{SmartJitter} [9]. This algorithm has three primary responsibilities: to initialize the map, to find the winning node for each input record, and to update the neighborhood about that node during each epoch. Although Fig. 3 does not do so, it is advisable to randomize the input order of the dataset because the input data may be potentially sorted, which can...
function FINDWINNER(\vec{I}, \Omega)
    \[ x_0, y_0 \leftarrow \text{premap}(i) \cdot [W_s, H_s] \]
    \[ x_e, y_e \leftarrow [x_0, y_0] + [W_s, H_s] \]
    \[ x_w, y_w, z_w \leftarrow \text{invalid} \]
    \[ d_{\text{min}} \leftarrow \infty \]
    \[ \text{for } 0 \leq z_i < D_s \text{ do} \]
    \[ \text{for } y_0 \leq y_i < y_0 \text{ do} \]
    \[ d_i \leftarrow || \vec{i} - \Omega x_i, y_i || \]
    \[ \text{if } d_i < d_{\text{min}} \text{ then} \]
    \[ d_{\text{min}} \leftarrow d_i \]
    \[ [x_w, y_w] \leftarrow [x_i, y_i] \]
    \[ \text{end if} \]
    \[ \text{end for} \]
    \[ \text{end for} \]
    \[ \text{return } [x_w, y_w, z_w] \]
end function

function TRAIN(I, \vec{O}, t_m)
    \[ [W, H] \leftarrow [W_p, W_s, H_p, H_s] \]
    \[ \rho_m \leftarrow \sqrt{W_pH_p} \]
    \[ \Omega \leftarrow \text{ALLOCMAP}(W, H, ||O||) \]
    \[ \text{INITWEIGHTVECTORS}(\Omega) \]
    \[ \text{for } 0 \leq t < t_m \text{ do} \]
    \[ \rho \leftarrow \rho_m \frac{t}{t_m} \]
    \[ \text{for all } i \in I \text{ do} \]
    \[ [x_w, y_w, z_w] \leftarrow \text{FINDWINNER}(\vec{i}, \Omega) \]
    \[ \text{UPDATENEIGHBORHOOD}(x_w, y_w, z_w, \rho, \lambda(t)) \]
    \[ \text{end for} \]
    \[ \text{end for} \]
end function

Fig. 2: The winning node for a record constrained to the “stack” of output nodes associated with its primary mapping (represented by the circle). The winning node is a node such that its distance between its weight vector and primary node is minimal. Note that \( H_p, W_p, \) and \( D_s \) can be set to virtually any positive, nonzero integer keeping in mind that large values associated with these variables could lead to memory-hungry topologies that can potentially exhaust the system resources.

Since we are extruding into the third dimension, then it would seem that a good starting point is to consider SmartJitter 2D. SmartJitter 2D already has one property that we’re interested in; primary mapping occurs in two dimensions. The secondary mapping, however, differs in that it is 3D. Thus, we now must consider a tertiary parameter for the secondary mapping, which we shall call \( D_s \). \( D_s \) is the depth of the secondary volume, and can be used to represent the size of the stack for each primary grid cell.

Fig. 4 demonstrates how SmartJitter 2D’s method for finding winning nodes is modified to deal with the newly-introduced third dimension. In SmartJitter 2D’s version of FINDWINNER, we constrain the search space to a rectangular area defined by its corners \([x_0, y_0]\) and \([x_e, y_e]\), which both lie on the same diagonal. In this case, \([x_0, y_0]\) represents the top-left corner and \([x_e, y_e]\) the bottom right; all \(x\) and \(y\) values to be inspected must fall into domains \([x_0, x_e]\) and \([y_0, y_e]\) respectively [28]. Because depth is not considered during primary mapping in SmartJitter Stacks, our search space is a volume \( W_s \times H_s \times D_s \), and we search the entire stack for winning nodes.

A couple of assumptions are made in this algorithm. First, it is expected that \( W_s, H_s \) and \( D_s \) are positive, nonzero integers. Zero magnitude in any of these dimensions is undesirable since it would mean that the search space would have zero volume, and we cannot map a record to nothing. As long as the above precondition is met, the winner coordinates (denoted by \([x_w, y_w, z_w]\)) will represent a valid output node in the map. It is also true that distance metrics other than the Euclidian Norm may be used, such as cosine distance coefficient and Pearson’s correlation coefficient [9].

A simplified implementation of UPDATENEIGHBORHOOD (which has been adapted to SmartJitter Stacks) is found...
Fig. 5: updateNeighborhood routine for SmartJitter Stacks.

in Fig. 5. This algorithm is similar to SmartJitter 3D’s updateNeighborhood routine [9], which must also update a volume in contrast to SmartJitter 2D’s grid of output nodes [28].

As an example, we present the 9,298-record Jiang-Rhoads dataset [26]. This dataset consists of gene expression data related to six developmental stages (embryo, larva1, larva2, larva3, larva4, adult) involving *Caenorhabditis elegans*, a nematode used as a common model organism for genetic and developmental research [15]. Additionally, it contains associated heavy and light polysomal gene expression data. A 2D projection of the dataset can be seen in Fig. 1 and an associated 3D Stacks projection in Fig. 6. Both plots, heavy polysomes (x-axis) are plotted against light polysomes (y-axis) in the primary scatter plot.

The Jiang-Rhoads dataset is a particularly useful demonstrator of SmartJitter Stacks because it has few dimensions which are suitable candidates for graph components. In cases of differential gene expression (DGE) experiments, the individual expression values for a gene are of less interest than the change in expression between samples. DGE can be particularly instructive, since regulatory changes in a few genes can lead to predominant changes in physiology [11]. Work by Golub, et al. has shown that DGE can be employed for differentiating two superficially similar classes of cancer for which classification is otherwise difficult and where proper classification is critical to successful treatment [11].

For this purpose, SmartJitter Stacks offers an advantage, in that it has the capability of creating an artificial dimension which can be used to detect differentiation in gene expression.

In order to achieve this in the Jiang-Rhoads dataset, we created a SmartJitter Stacks SOM such that $[W_p, H_p] = [100, 100]$ and $[W_s, H_s, D_s] = [1, 1, 50]$. This permits competition in only the new $z$ dimension.

Fig. 7 demonstrates an important property of SmartJitter Stacks: records within a visual cluster have similar profiles with respect to the SOM’s organized dimensions. This permits the rapid identification of similar records purely through visual means. If we create an associated parallel coordinates plot for the records in a cluster displaying magnitudes for each organized dimension, then we find that the profiles and values of the records are similar, despite the fact that the values for one unorganized dimension (polysomal class) are notably different.

IV. PERFORMANCE EVALUATION

By extending SmartJitter into the third dimension, we increase both the running time and resources utilized by SmartJitter. SmartJitter Stacks shares the same kind of upper bounds on computational complexity as SmartJitter 3D. If each of the dimensions and the cardinality of the organized dimension set are equal to $n$, then we find that the upper bound is $O(n^4)$, since SmartJitter Stacks has to produce a volume of vectors. Similarly, the neighborhoodUpdate portion of SmartJitter requires iterating over a volume of output nodes, and then performing vector operations on each. However, although most of the time is spent in neighborhoodUpdate, in order to address the time complexity of SmartJitter Stacks, we must also consider some other properties related to the training algorithm as a whole.
One of the contributors to the time complexity of the training algorithm is the iteration over each record at least $t_m$ times (Fig. 3). Since neighborhood radius decreases over time we are shrinking the search space for neighborhoodUpdate in subsequent epochs. Therefore, the time complexity for a dataset with $m$ records is more properly modelled as $O(n^4m \int_{t_0}^{t_m} \rho_m^{2/4n} \frac{\rho_m^{2/4n}}{t_m} dt)$.

SmartJitter Stacks’ performance scales poorly and this can limit its utility in visualization tasks. High-resolution SOMs are often desirable with respect to knowledge discovery because they provide the opportunity for two similar (yet distinct) records to be mapped to different output nodes; however, higher-resolution SOMs imply a larger search space and therefore greater time requirements and demand for system resources such as memory.

Recent developments in end-user computer hardware have made parallel computing readily available. Multicore computing and symmetric multiprocessing are ubiquitous in modern personal computers, and operating system support for these features is well established [8]. Research environments with relatively new workstations can thus provide access to tightly-coupled parallel environments. Additionally, the lowering costs associated with the PC architecture has permitted greater access to computational clusters, which provide a scalable and distributed environment for parallel computing [5]. Finally, the advent of general purpose computation on the GPU (GPGPU) has provided a very tightly-coupled computational environment affording hundreds or thousands of hardware threads at low cost [1], [16], [24]. For these reasons, parallelism provides an attractive opportunity for time-optimization of SmartJitter Stacks.

While parallelism is an attractive prospect in this case, the degree to which it is effective in decreasing running time depends strongly on the application in question. An objective measurement of this decrease in running time, or speedup, can be stated as the ratio of the “optimized” process’s running time to that of its unoptimized counterpart. Via Amdahl’s Law (6), one can predict the speedup of a parallel process compared to its serial counterpart [2]:

$$S_a(N, P) = \frac{1}{(1 - P) + P/N} \quad (6)$$

In (6), $P$ denotes the percentage of code which is in a parallel region (i.e., coverage), and $N$ refers to the number of threads assigned to the process. One finds that in most cases, Amdahl’s law converges towards an asymptote, as illustrated in (7):

$$\lim_{N \to \infty} S_a(N, P) = \frac{1}{(1 - P)} \quad (7)$$

In one case ($P = 1$), we find that Amdahl’s law is actually divergent, yielding what is known as linear speedup. However, it is very rare for this circumstance to actually occur. Access to shared resources (even at the hardware level) usually implies critical sections; otherwise, data hazards may occur [27]. It may also be necessary to impose other synchronization primitives such as barriers in order to properly carry out a
calculation [3]. These events effectively serialize the process while they are being carried out, so it is exceedingly unlikely that one will achieve \( P = 1 \).

However, it is worth noting that, while \( P = 1 \) is an extremely rare condition, \( P \approx 1 \) is certainly possible and that such \( P \) can produce results approaching linear speedup. Note that in Fig. 8, \( P = 0.5 \) yields a maximum speedup of 2; however, \( P = 0.75 \) doubles that limit and \( P = 0.999 \) exhibits effective linear speedup for \( N < 2^8 \). High-coverage code results in what is sometimes called an *embarrassingly parallel* implementation. If SmartJitter Stacks can be implemented in a manner which is embarrassingly parallel, then it can take advantage of the environments previously enumerated, and become more useful as a result.

There is some evidence to support the idea that SOMs and their brethren can be implemented in an embarrassingly parallel fashion. Weitzenfeld, et al., for instance, were able to identify neurons as archetypical objects and applied concurrency to a neural network simulator [29]. Lawrence, et al. later conducted research showing that SOMs can be implemented in such a way as to facilitate high coverage, although the performance of the map is sensitive to the nature of the input data [19]. McConnell, et al. have implemented a SOM in a CUDA environment, noting good scalability [21].

As starting points for parallelizing SmartJitter Stacks, we looked towards the findWinner and updateNeighborhood routines (Fig. 4, Fig. 5). As Tanenbaum points out, the most effective time optimizations tend to target code which is executed throughout the majority of a program’s running time [27]. Although some profiling was necessary to confirm this, it is intuitive that these routines comprise the majority of SmartJitter Stacks’ running time. In order to test this hypothesis, we collected a sample of 32 instances of SmartJitter Stacks utilizing Fisher’s Iris data [10] and a SOM with primary grid 100 \( \times \) 100 and secondary grid 10 \( \times \) 10 \( \times \) 10, which each ran for a period of 1000 epochs. We found that the total time spent in either region was about 99.98% of the program’s running time, with updateNeighborhood composing the bulk of this time (Fig. 9).

Given the above results, it is conceivable that by properly parallelizing findWinner and updateNeighborhood, near-linear speedup may occur. Let \( T \) be the total execution time of SmartJitter Stacks. A simple model of \( T \) is the sum of the times of its constituent parts, such that \( T = T_s + T_p \), where \( T_s \) denotes time spent in serial regions and \( T_p \) is time in parallel regions. From the above experiment, we hypothesize that \( T_s << T_p \). We can model \( T_p \) such that \( T_p = T_{fw} + T_{nhu} \), where \( T_{fw} \) and \( T_{nhu} \) represent total times spent in the findWinner and neighborhoodUpdate regions respectively. To start, we can model \( T_{fw} \) as in (8):

\[
T_{fw} \propto \int m(W_s H_s D_s) dt_m = t_m \cdot m(W_s H_s D_s) + c \quad (8)
\]

Yet, \( T_{nhu} \) is slightly more difficult to model since its search space is a function of time and shrinking with each successive epoch. We then express the time spent for updateNeighborhood in a single epoch as in (9):

\[
T_{nhu}(t) \propto m(WHD_s) \rho(t) \quad (9)
\]

With (9), one can make a more generalized statement about the total running time required by updateNeighborhood for the entire process. This prediction can be achieved via the definite integration of (9) with respect to time, as shown in (10):

\[
T_{nhu}(t) \propto m(WHD_s) \int_{t_m}^{t_f} \rho(t) dt + m(WHD_s) \frac{t_m^2 \rho(t_m) \rho(t_f)}{ln(\rho(t_f))} + c \quad (10)
\]
With this information in hand, it is possible to predict how altering various parameters for SmartJitter Stacks might affect coverage, as in Fig. 10. It is perhaps intuitively obvious that the total number of epochs would affect coverage, as the running time is directly proportional to the number of epochs specified (Fig. 3). Record count may also have an effect on running time, for fewer records mean that a single epoch lasts for a shorter period of time. In general, one predicts that a greater workload serves to increase the coverage of SmartJitter Stacks, therefore increasing the upper bound for $S_a$.

V. EXPERIMENTAL DATA

In order to test the efficacy of our parallelization strategy, we designed an experiment to allow each of the above parameters to be adjusted in isolation. We collected data from a total of 12 samples, whose sizes are illustrated in Table I. Half of the samples that were collected were derived from Fisher’s Iris dataset [10], while the remainder were derived from the Jiang-Rhoads dataset. In addition to collecting data for the serial implementations of SmartJitter Stacks, we also collected data for each of the specified processes using an MPI-based implementation. For our MPI implementation, samples utilizing 2 and 4 threads were both collected. The times recorded for each sample start from the beginning of training to its end, and thus excludes steps such as reading in the source dataset.

The computational cluster used for the experiment is based on a 312-core Apple Xserve cluster running GNU/Linux. Each node provides 16 CPU cores, 24GB of RAM on the head node, and 15GB of RAM on each compute node. The cluster utilizes Gigabit Ethernet for the internode communication. Process scheduling is provided via Sun Grid Engine [25] and MPI support via MPICH2 [4].

While there may be minor differences between the two SmartJitter implementations that are difficult to reconcile, we made several efforts to ensure that neither implementation had an artificial advantage over the other. For instance, one potential bottleneck in the MPI implementation is the communications interconnect, which has considerably less bandwidth and greater latency than the memory bus used by the CPU [14]. A naïve MPI implementation with a considerable amount of message passing might be limited by this situation, so the solution to this was to reduce message passing as much as possible, such that the memory bus and similar high-throughput links are utilized extensively by the program. We achieved this largely through caching. Each process recieves its own copy of the input dataset and broadcast communications are used whenever possible, since they can be performed in logarithmic time [3]. In the early development of our program, we found that this approach considerably reduced communication overhead.

One thing that was also considered was the division of work between threads. We sought a partitioning scheme that would provide increased throughput, such that the largest number of output nodes possible is processed between synchronization points. Work division also has implications from the perspective of memory layout, since there is the possibility that each thread will be assigned its own private address space.

After close inspection of the code in Fig. 4 and Fig. 5, we chose an approach where (for $N$ threads) we partition the data according to the dimension $d_i$ in the vector $[D, H, W]$ such that (compared to some $d_j$) $d_i \geq N \land i < j$. This method was largely inspired by the NTSC interlacing, and results in three possible strategies for work division: interleaved, interlace-by-row, and interlace-by-plane. Thus, the optimal partition is chosen as described in (11):

$$p(W, H, D, N) = \begin{cases} D \geq N : & (\text{Interlace by plane}) \\ H \geq N : & (\text{Interlace by row}) \\ else : & (\text{Interleave}) \end{cases}$$

Note that in the case where $W \times H \times D < N$, not all threads are assigned work and some threads must sit idle. In such a case, we may find that the cost of synchronization overhead is so great that a parallel implementation of SmartJitter Stacks may be slower than its corresponding serial implementation. However, it is also worth noting that as a volume, the search space can scale dramatically with the increase of a single parameter, and in the case where all dimensions are of equal value, it scales cubically.

The goal of this partitioning scheme is to supply as much input to the $\text{findWinner}$ and $\text{updateNeighborhood}$ kernels as possible, such that much of the time in the kernel is spent executing a large section of the loops in Fig. 4 and Fig. 5. This means that synchronization conditions, such as barriers,
TABLE II: Average Speedup

<table>
<thead>
<tr>
<th>Process</th>
<th>MPI-2</th>
<th>MPI-4</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10x10, 10x10x10, E=1000</td>
<td>7.00</td>
<td>10.44</td>
<td>Iris</td>
</tr>
<tr>
<td>10x10, 10x10x10, E=10000</td>
<td>59.17</td>
<td>109.80</td>
<td>Iris</td>
</tr>
<tr>
<td>10x10, 10x10x10, E=1000</td>
<td>8.00</td>
<td>15.57</td>
<td>Jiang Rhoads</td>
</tr>
<tr>
<td>10x10, 10x10x10, E=10000</td>
<td>68.14</td>
<td>138.04</td>
<td>Jiang Rhoads</td>
</tr>
</tbody>
</table>

occur less frequently. Thus, such a partitioning scheme should result in better coverage.

The results for the experiment are listed in Table II. We generally found that speedup was superlinear. (i.e., $S_A > N$) While this is somewhat surprising, it is possible that parallelization has improved cache utilization in this instance. Effective cache utilization has been known to result in super-linear speedup and, in a distributed environment, cache misses often decrease with the number of hardware threads available [7].

VI. CONCLUSIONS

We have presented a method of applying self-organization to a two-dimensional scatter plot which extends the data into an artificial third dimension based on record distance. The primary advantage of SmartJitter Stacks is in easier identification of records with similar profiles (as compared to SmartJitter 2D) without sacrificing the insight that can follow from a traditional scatter plot. Because of this, we hypothesize that users may leverage SmartJitter Stacks to obtain better insight into the datasets they are interested in.

We have also found that SmartJitter Stacks is a good candidate for parallelization. This shows that although the serial version of SmartJitter Stacks does not scale well, it is possible to construct an implementation of SmartJitter Stacks with high speedup for large datasets $N$.

VII. FUTURE WORK

One thing that is curious is the superlinear speedup associated with the MPI implementation of SmartJitter Stacks. It is interesting to see if this can be replicated in other experiments. In order to test this, it may be useful to implement SmartJitter in another environment, such as in a GPGPU environment, since GPUs are generally massively parallel and provide global, thread-block local, and thread-local memory. Furthermore, some synchronization overhead may be avoided since kernels run in lockstep [1], [16], [24]. This provides one with an opportunity to test the cache utilization hypothesis mentioned in the "Parallelism Experiment" section. Thus, one thing that we plan to do is to implement this for a GPGPU environment.

Additionally, we intend to conduct a formal usability study to evaluate the effectiveness of SmartJitter Stacks in a variety of use cases. The Jiang-Rhoads dataset responded favorably to SmartJitter Stacks, despite its density. We expect a similar outcome from processing related datasets.

ACKNOWLEDGEMENTS

This project was supported by grants from the National Center for Research Resources (5P20RR016456-11 and 5P20RR018724-10) and the National Institute of General Medical Sciences (8 P20GM103424-11 and 8 P20 GM103433-10) from the National Institutes of Health. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of NIH. Authors also wish to thank John Fisher for help with computational clusters and related hardware.

REFERENCES


